VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Implementace neuronové sítě na platformě CUDA

Implementation of Neural Network on CUDA Platform

2019 Tomáš Chovančík

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Tomáš Chovančík**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Implementace neuronové sítě na platformě CUDA
Implementation of Neural Network on CUDA Platform

Jazyk vypracování: čeština

Zásady pro vypracování:

Cílem práce je implementace neuronové sítě a algoritmu jejího učení v jazyce C++ a jeho následná paralelní implementace pomocí technologie CUDA.

Během řešení postupujte dle následujících bodů:
1. Popište problematiku neuronových sítí a jejich učení.
2. Popište technologii CUDA.
3. Nastudujte a popište existující implementace neuronových sítí na platformě CUDA.
4. Implementujte a otestujte neuronovou síť na vzorových datech.
5. V závěru zhodnoťte dosažené výsledky.

Seznam doporučené odborné literatury:

[1] Beale, Russell, and Tom Jackson. Neural Computing-an introduction. CRC Press, 1990.
[2] Foster, Ian. "Designing and building parallel programs." (1995).
[3] Nvidia, C. U. D. A. "Programming guide." (2008).
[4] Chetlur, Sharan, et al. "cudnn: Efficient primitives for deep learning." arXiv preprint arXiv:1410.0759 (2014).

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Lumír Kojecký**

Datum zadání: 01.09.2017

Datum odevzdání: 30.04.2019

VYSOKÁ ŠKOLA BÁŇSKÁ-TECHNICKÁ UNIVERZITA OSTRAVA Fakulta elektrotechniky a informatiky

doc. Ing. Jan Platoš, Ph.D.
vedoucí katedry

prof. Ing. Pavel Brandštetter, CSc.
děkan fakulty

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 30. dubna 2019 .....

Rád bych zde poděkoval vedoucímu své bakalářské práce Ing. Lumíru Kojeckému za odborné vedení práce, přínosné a cenné rady, připomínky i konzultace, které vedly k vytvoření této práce.

Abstrakt

Bakalářská práce popisuje sekvenční implementaci vícevrstvé perceptronové neuronové sítě s dopředným šířením a algoritmus rojení částic pro její učení a jejich následnou paralelní implementaci na CUDA platformě v jazyce C++. V práci je věnována pozornost teoretické problematice neuronových sítí, evolučních algoritmů a paralelní platformy CUDA. Následuje samotný popis a návrh obou implementací. Na konci práce jsou provedeny jednotlivé testy neuronové sítě na XOR problému, aproximaci funkce sinus a vzorových datech.

Klíčová slova: neuronové sítě, vícevrstvé sítě, neuronové sítě s dopředným šířením, neuron, perceptron, klasifikace, evoluční algoritmy, rojení částic, CUDA

Abstract

Bachelor thesis deals with a serial implementation multi-layer perceptron's feedforward neural network and evolution algorithm of particle swarm for its learning and their following parallel implementation on the CUDA platform in C++. The thesis focus on teoretical problematics of neural network, evolutionary algorithms and parallel platform of CUDA. Then follows the description and design of both implementations. At the end of the thesis individual tests of neural network on XOR problem, approximation sine function and sample data are performed.

Key Words: neural networks, multi-layer networks, feedforward neural networks, neuron, perceptron, classification, evolution algorithm, particle swarm, CUDA

Obsah

Seznam použitých zkratek a symbolů

CPU	–	Central Processing Unit
CSV	–	Comma-separated values
CUDA	–	Compute Unified Device Architecture
DRAM	–	Dynamic Random Access Memory
GPU	–	Graphic Processing Unit
HTML	–	Hypertext Markup Language
PSO	–	Particle Swarm Optimization
SIMD	–	Single instruction, multiple data
SIMT	–	Single instruction, multiple thread
SOMA	–	Self-Organizing Migrating Algorithm
TPU	–	Tensor Processing Units
XML	–	Extensible Markup Language
XOR	–	Exclusive disjunction

Seznam obrázků

Seznam tabulek

Seznam výpisů zdrojového kódu

1 Úvod

Cílem této práce je implementovat neuronovou síť a algoritmus učení v jazyce C++ sekvenčně a následně paralelně pro použití na grafických kartách s platformou CUDA. Na základě domluvy s vedoucím této práce byla zvolena implementace vícevrstvé sítě s dopředným šířením a pro její učení byl vybrán evoluční algoritmus rojení částic. Následně si rozebereme testování obou implementací na klasifikačních úlohách a jedné aproximační úloze. Přesněji na problému XOR, aproximaci sinus funkce a vzorových datech. Porovnány budou z hlediska průměrné doby učení.

Tato práce je rozdělena do tří části: teoretické, popisu implementace a testování v celkem šesti kapitolách.

První kapitola bude pojednávat o neuronových sítích a jejich učení. Popíšeme si zde neuronovou síť, biologický a umělý neuron, perceptron nebo také adaptace perceptronu pomocí Hebbova pravidla, učení pomocí metody zpětného šíření v síti, nazývaného angl. backpropagation a použití evolučních algoritmů k učení sítě. Dále si projdeme různé typy neuronových sítí, kterými jsou vícevrstvé dopředné sítě a jejich upravené verze, jako je rekurentní vícevrstvá síť nebo hluboké vícevrstvé neuronové sítě a také ostatní typy např. Kohonenovy samoorganizující se sítě, Hopfieldova síť.

Druhá kapitola se bude zabývat evolučními algoritmy. Bude zde popsán jedinec, populace, účelová funkce, testovací funkce a princip a popis algoritmu rojení částic. Dále si zde popíšeme generátory pseudonáhodných čísel Mersenne Twister a logistickou rovnici.

Ve třetí kapitole bude věnována pozornost CUDA platformě a její architektuře z pohledu hardwarového i programátorského. Objeví se zde stručný popis SIMD a SIMT architektury, vybraných nejznámějších knihoven a frameworků jako jsou Caffe, cuDNN, TensorFlow vyvíjené pro řešení problematiky s učením neuronových sítí a již publikované projekty nebo články zabývající se tímto problémem na grafických kartách s CUDA.

U čtvrté kapitoly probereme uživatelské rozhraní a hlavně návrh a jednotlivé třídy sekvenční a paralelní CUDA implementace neuronové sítě a algoritmu rojení částic použitého k jejímu učení. Dále také implementace a zařazení generátorů pseudonáhodných čísel.

V páté kapitole si otestujeme samotný algoritmus PSO z pohledu funkčnosti na testovacích funkcích. Obě implementace pak ještě budou porovnány jednotlivě na XOR problému, aproximaci sinus funkce a vzorových testovacích datech z hlediska času učení na procesoru a grafické kartě při různě nastavených parametrech.

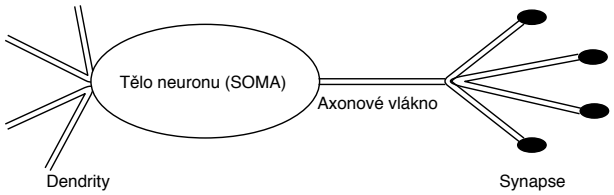
2 Neuronové sítě

Umělé neuronové sítě vycházejí z biologických neuronových sítí a z hlediska principu funkcionality je kladena snaha, aby byly podobné své předloze [1]. Samotný biologický vzor [1, 2] se skládá z neuronů, které jsou navzájem propojeny pomocí axonových vláken, přenášející výstup neuronu k synapsím. Synapse [1] jsou pak výstupem neuronu předávající signál dalším neuronům na vstup nazývaný dendrity.

U neuronových sítí považujeme za důležitou vlastnost schopnost se učit z předkládaných vzorů, které se vyskytují v tzv. trénovací sadě vzorů a dohromady představují aktuálně řešený problém. Naučené vzory jsou rozloženy v celé síti ve formě synaptických vah. Synaptické váhy pak představují hodnoty určující směr průchodu signálu sítí až k výsledku. Toho je dosaženo pomocí oslabování či zesilování těchto hodnot. [1]

2.1 Neuron

Hlavním prvkem umělých neuronových sítí je umělý neuron inspirovaný biologickým neuronem [1]. Funkci biologického neuronu [1, 2] jsme si ve své podstatě popsali již v předešlé podkapitole 2. Podrobněji si ji můžeme vysvětlit následovně: na vstupy (dendrity) jsou přivedeny vstupní signály a v těle neuronu, označovaném také jako soma, jsou sečteny. Pokud součet nepřesáhne prahovou hodnotu je neuron neaktivní a žádný výstup nevrací. V opačném případě je výstup dále poslán přes axonové vlákno na synapse zesilující nebo oslabující výstupní signál. Synapse [1] jsou pak připojeny k dendritům dalšího neuronu. Jednoduchého schématu biologického neuronu si můžeme všimnout na následujícím obrázku 1, tento obrázek byl inspirován schématy z literatury [1, 2].



Obrázek 1: Schéma biologického neuronu.

2.2 Typy neuronů

Existují také různé typy neuronů. Mezi ty nejznámější můžeme zařadit např. McCullochův model neuronů nebo také Perceptron. [1]

2.2.1 Perceptron

Tento typ umělého neuronu byl navržen Frakem Rosenblattem v roce 1962 a patří mezi nejdůležitější modely [1, 2]. Vychází ze základního neuronu a lze jej definovat jako součet součinů

vstupů s váhami, jehož výsledek je pak porovnán s prahovou hodnotou daného neuronu. V případě, že je výsledná hodnota menší než prahová, výstupní hodnotou je nula. Pokud je větší než prahová hodnota, neuron je aktivován a výsledná hodnota je jedna. [2]

Předchozí popis vidíme na následujících vztazích převzatých z literatury [1], první vztah 1 je vážený součet vstupů a druhý vztah 2 prahová funkce:

$$y = \sum_{i=1}^{N} w_i x_i \tag{1}$$

kde,
N – počet vstupů,
w – váha,
x – vstup.

$$y = Signum\left(\sum_{i=1}^{N} w_i x_i - \vartheta\right) \tag{2}$$

kde,
ϑ – prahová hodnota neuronu,
$Signum(x) = 1, x > 0$
$Signum(x) = 0, x \leq 0$

2.3 Typy neuronových sítí

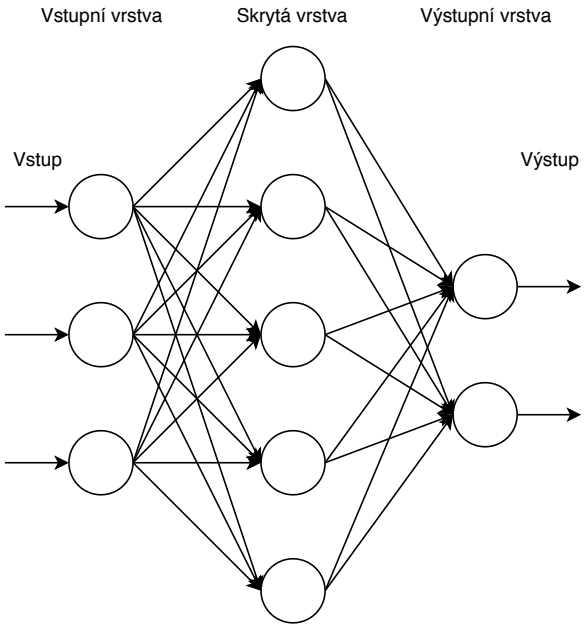
Neuronové sítě se nacházejí v různých formách pro různá využití. V následujících podkapitolách budou popsány vybrané typy, především vícevrstvé sítě s dopředným šířením, rekurentní vícevrstvé sítě a hluboké neuronové sítě. V závěru kapitoly se budeme věnovat Kohonenovým samoorganizujícím se sítím a Hopfieldovým sítím.

2.3.1 Vícevrstvé perceptronové sítě

Tento typ sítí je založen na principu [2] navzájem propojených vrstev neuronů, konkrétně perceptronů. Skládá se z několika vrstev: ze vstupní, alespoň z jedné skryté vrstvy a výstupní vrstvy. Perceptrony [2] se nacházejí ve skryté resp. skrytých vrstvách a výstupní vrstvě. Vrstvy jsou pak propojeny tak, že výstupy předchozí vrstvy jsou připojeny na vstupy následující vrstvy [2, 1].

Šíření informace v sítí pak probíhá podle tzv. dopředného šíření [1] (angl. feed forward), kdy na vstupy dosadíme nějaké hodnoty, tyto vstupy jsou následně upraveny synaptickými váhami a poslány jako vstupy do následující vrstvy, kde se hodnoty sečtou podle vztahu 1 a projdou aktivační funkcí, která bude popsána v následující podkapitole 2.3.2. Celý průběh se pak opakuje až do výstupní vrstvy, z jejichž výstupů získáme výstupní hodnoty celé sítě. Vstupy a výstupy neuronové sítě jsou organizovány v trénovací sadě vzorů jako dvojice vstupů a výstupů postupně seřazených za sebou. [1]

Na obrázku 2 můžeme vidět schéma vícevrstvé neuronové sítě s dopředným šířením, tento obrázek je inspirován literaturami [1, 2].



Obrázek 2: Příklad vícevrstvé perceptronové neuronové sítě.

2.3.2 Aktivační funkce

Tyto nelineární funkce [2] jsou využívány především u vícevrstvých neuronových sítí a na rozdíl od prahové funkce je jejich výstup reálné číslo. Udávají tedy přesnější informace pro případné učení sítě [2]. Existuje více druhů aktivačních funkcí, zde si pro příklad popíšeme pouze tyto následující funkce, vzhledem k tomu, že budou použity v implementaci.

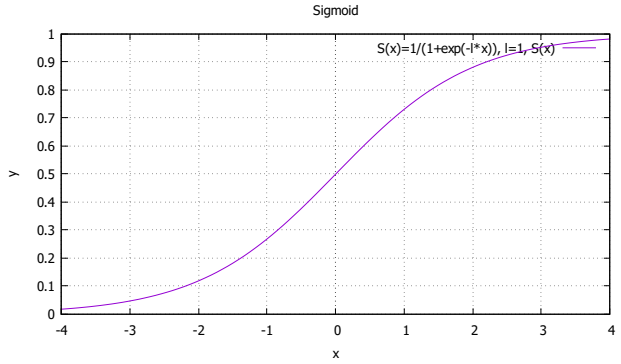
2.3.2.1 Sigmoid Nelineární aktivační funkce, nejčastěji využívaná v neuronových sítích s učením pomocí algoritmu zpětného šíření (angl. Backpropagation) [3]. Funkce je definována vztahem 3 v oboru hodnot $\langle 0, 1\rangle$ podle literatury [1]. Pro ilustraci můžeme funkci vidět v grafu [1] na obrázku 3.

$$S(y) = \frac{1}{1 + e^{-\lambda y}} \tag{3}$$

kde, λ je konstanta udávající strmost Sigmoid funkce podle literatury [1]. V článku [3] ale tato konstanta zmiňována není [2]. y je výsledná hodnota vztahu 1 a $S(y)$ je aktivační funkce Sigmoid.

[1] Vygenerován pomocí nástroje gnuplot dostupného na webové adrese: `http://www.gnuplot.info/`.

[2] Z tohoto důvodu bude v implementaci konstanta λ nastavena na hodnotu 1, při této hodnotě nebude ovlivňovat výsledek funkce.

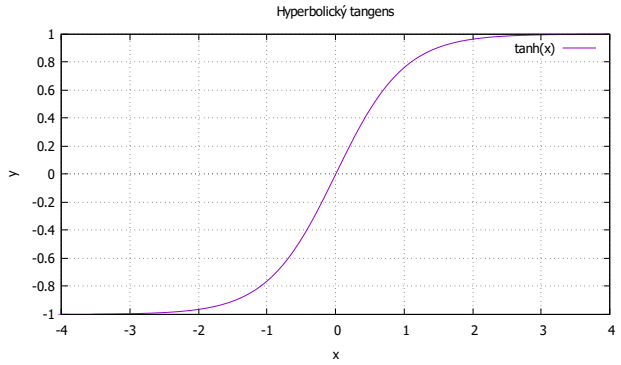


Obrázek 3: Graf aktivační funkce Sigmoid.

2.3.2.2 Hyperbolický tangens Tato nelineární funkce je definována v oboru hodnot $\langle -1, 1 \rangle$ vztahem 4 pocházejícím ze článku [3]. Graf [3] funkce je znázorněn na obrázku 4.

$$Tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{4}$$

kde, x je výsledná hodnota vztahu 1 a $Tanh(x)$ je aktivační funkce Hyperbolický tangens.



Obrázek 4: Graf aktivační funkce Hyperbolický tangens.

2.3.3 Rekurentní vícevrstvé sítě

Jedná se o upravené vícevrstvé neuronové sítě, skládající se z tzv. rekurentních neuronů [1]. Tyto neurony [1] jsou napojeny na vstupy předcházející vrstvy neuronů a jejich vstupem je výstup konkrétního neuronu v následující vrstvě. Každý z těchto rekurentních neuronů předchozí vrstvy má na svém vstupu pouze jeden výstup z konkrétního neuronu v následující vrstvě. Tento typ sítě má při každém průchodu sítí jinou odezvu [1].

[3] Vygenerován pomocí nástroje gnuplot dostupného na webové adrese: `http://www.gnuplot.info/`.

2.3.4 Hluboké vícevrstvé dopředné neuronové sítě

Oproti běžným vícevrstvým neuronovým sítím mají mnohem více skrytých vrstev a využití nacházejí především pro řešení komplexnějších problémů. [4, 13]

2.3.5 Kohonenovy samoorganizující se sítě

Typ sítí navržených profesorem Kohonenem, kde jsou sítě založeny na principu samoorganizace a adaptivního učení, tj. že neuronové sítě jsou schopny se samy organizovat a přizpůsobovat [2].

Kohonenovy mapy [2] jsou na rozdíl od ostatních typů sítí složeny jen ze vstupní vrstvy a do mřížky organizované výstupní vrstvy. Každý vstup je pak propojen s každým uzlem ve výstupní vrstvě a ty jsou pak propojeny pouze ze sousedními uzly v rámci vrstvy.

2.3.6 Hopfieldova síť

Tento typ sítě byl navržen a vytvořen Johnem Hopfieldem v 80 letech 20. století [2, 1]. Z pohledu návrhu [1] jsou všechny neurony v této síti navzájem propojeny vazbami. Jednotlivé neurony jsou ve své podstatě hodně podobné perceptronům, jako vstup berou výstupy všech okolních neuronů, které jsou potom přivedeny do váženého součtu s váhami sítě. Mají i prahovou hodnotu a aktivační funkci, která bere tuto sumu jako vstup[2, 1]. Neurony lze u těchto sítí rozdělit na [1, 2]:

- **binární** – (označované také jako standardní v [1]) nabývají hodnot 0 nebo 1.
- **bipolární** – nabývají hodnot -1 nebo $+1$.

Tyto sítě využívají zásadně bipolárních stavů neuronů [1]. Principem je model fungující v cyklech, kdy se jednotlivé neurony mezi sebou navzájem ovlivňují. Po nastavení jednoho vstupu v konkrétním neuronu, je vypočten výstup a předán na vstup ostatním neuronům. Takto se pak dále postupuje cyklicky dokud není síť stabilní.[1]

K trénování těchto sítí na vzorech je využita tzv. energetická funkce [1, 2]. Jejím cílem je dojít k nalezení lokálního minima v energetické oblasti, kde minimum představuje stabilní řešení.

V těchto sítích nastává i možnost, kdy může dojít k tzv. uváznutí v nežádoucím stavu. Toto řeší tzv. Boltzmannův stroj, který je dále popsán v literatuře [1].

2.4 Metody učení neuronových sítí

Neuronové sítě můžeme učit hned několika způsoby či metodami. Některé tyto metody se používají jen pro konkrétní typy sítí. Příkladem těchto metod mohou být následující zmíněné.

2.4.1 Hebbovo učení

Hebovo učení bylo navrženo v roce 1949 Donaldem Hebbem [1, 2] a slouží k naučení neuronu typu perceptron. Princip algoritmu [1] je v nastavení vah a prahové hodnoty náhodnými čísly, výpočet

aktuálních výstupů na základě vloženého trénovacího vzoru a samotná adaptace perceptronu, tedy úprav jeho vah.

2.4.2 Metoda Backpropagation

Učení neuronové sítě Backpropagation je založeno, jak nám už z názvu vyplývá, na základě zpětného šíření v síti [1].

Z počátku se vyhodnotí první vzor trénovací sady v celé síti. Od aktuálních výstupů (označovány také jako odezvy [1]) této sítě jsou odečteny požadované výstupy. Tento rozdíl určuje chybu v neuronové síti, ta pak spolu s hodnotou *learning rate* upravuje váhy sítě zpětně od posledních vrstev k prvním pro snížení chyby v následujících odezvách. Učení probíhá pro každý vzor v trénovací sadě. [1]

Výpočet chyby v neuronové síti si můžeme vyjádřit následujícím vztahem, který je převzat z literatury [1]:

$$E = \frac{1}{2} \sum_{i=1}^{p} \sum_{j=1}^{m} (y_j - o_j)_i^2 \tag{5}$$

kde,
y_j – aktuální výstupy vrstvy,
o_j – požadované výstupy,
m – počet výstupů,
p – počet vzorů v trénovací sadě.

2.4.3 Učení pomocí evolučních algoritmů

Pro vícevrstvé dopředné neuronové sítě se používají i jiné metody učení než je např. Backpropagation [1] z předchozí podkapitoly 2.4.2. V oblasti paralelního zpracování by byla tato zmiňovaná metoda obtížněji implementovatelná už z principu jejího fungování. Proto se naskytuje možnost použití [6, str. 46] evolučních algoritmů, které jsou lépe paralelizovatelné.

Učení pomocí evolučního algoritmu, resp. optimalizační techniky rojení částic je popsáno v článku [5]. Samotný návrh pro učení neuronové sítě podle článku [5] je takový, že jsou jednotlivé váhy neuronové sítě vytyčeny jako oblast hledání problému, nebo-li také dimenze příslušného evolučního algoritmu.

3 Evoluční algoritmy

Jedná se o specifickou skupinu algoritmů – optimalizačních technik, inspirující se v základu evoluční teorie. [6]

Tyto algoritmy jsou založeny na populaci jedinců v definovaném prostoru, jejichž úkolem je vytváření nových jedinců, tzv. potomků a následné uvolnění životního prostoru pro nové nebo lepší jedince. Hlavním cílem těchto optimalizačních technik je tedy dojít k optimálnímu výsledku nalezením minima či maxima určené funkce, která je obecně nazývána jako účelová funkce. Mezi optimalizační techniky můžeme zařadit algoritmy, jako rojení částic, SOMA, diferenciální evoluce nebo také rozptýlené hledání a další. [6]

3.1 Základní pojmy

Evoluční optimalizační techniky definují hned několik pojmů, z nichž nejdůležitější jsou níže popsané.

3.1.1 Jedinec

Jedinec [6] představuje řešení nějakého stanoveného problému. Taktéž je svázán s výslednou hodnotou stanovené účelové funkce – vhodností. Ta pak obsahuje jen informaci představující kvalitu daného jedince. [6]

Existují různé typy jedinců [6]:

- **binární** – tvořen pouze sekvencí 0 a 1, tzv. chromozom.
- **reálná/celočíselná** – tvořen čísly reálnými, celými nebo jejich kombinacemi.
- **nenumerické** – nenumerické hodnoty.
- **strom** – „umožňuje vizualizovat jedince jako stromovou strukturu“ [6, str. 96].

3.1.2 Populace

Skupina, nebo-li množina [6] jedinců, z nichž každý jedinec je řešením nějakého konkrétního problému. Použití populace je typické pro evoluční algoritmy.

V principu [6] dochází v průběhu času k cyklickému vzniku nových populací na místo starých populací, které po nahrazení zaniknou.

Druh evolučního algoritmu můžeme určit podle matematických pravidel, chování nebo typu jedinců [6].

3.1.3 Účelová funkce

Účelová funkce [6] $f(x)$ nebo $f_{cost}(x)$, anglicky „cost function“, je optimalizační funkce, kterou můžeme vidět i jako geometrický problém, ve kterém je hledáno maximum nebo minimum pro získání optimální hodnoty, na základě argumentů funkce. V některých algoritmech jsou argumenty označovány jako pozice jedince v prostoru. Výsledná hodnota je označována někdy i jako vhodnost (angl. fitness), přesněji řečeno, je to upravená výsledná hodnota. Označení využíváme v případech, kdy je hledána maximální výsledná hodnota. [6]

3.1.4 Testovací funkce

Tyto funkce slouží pro testování evolučních algoritmů. Známe také jejich globální extrém pro určitou kombinaci jejich parametrů. [6]

Vybrané testovací funkce použité v této práci jsou převzaty z literatury [6] (N značí dimenzi řešeného problému, resp. počet vstupních argumentů funkce):

- **1. De Jong funkce** – pro $(x_1,\ x_2, \ldots,\ x_N) = (0,\ 0, \ldots,\ 0)$, je její globální minimum $y = 0 \cdot N$. Funkci lze definovat následujícím vztahem:

$$\sum_{i=1}^{N} x_i^2 \tag{6}$$

- **2. De Jong funkce** – pro $(x_1,\ x_2, \ldots,\ x_N) = (1,\ 1, \ldots,\ 1)$, je její globální minimum $y = 0 \cdot N$. Funkci lze definovat následujícím vztahem:

$$\sum_{i=1}^{N} 100\left(x_i^2 - x_{i+1}\right)^2 + (1 - x_i)^2 \tag{7}$$

- **3. De Jong funkce** – pro $(x_1,\ x_2, \ldots,\ x_N) = (0,\ 0, \ldots,\ 0)$, je její globální minimum $y = 0 \cdot N$. Funkci lze definovat následujícím vztahem:

$$\sum_{i=1}^{N} |x_i| \tag{8}$$

- **4. De Jong funkce** – pro $(x_1,\ x_2, \ldots,\ x_N) = (0,\ 0, \ldots,\ 0)$, je její globální minimum $y = 0 \cdot N$. Funkci lze definovat následujícím vztahem:

$$\sum_{i=1}^{N} i x_i^4 \tag{9}$$

- **Schwefelova funkce** – pro $(x_1,\ x_2, \ldots,\ x_N) = (420,969,\ 420,969, \ldots,\ 420,969)$, je její globální minimum $y = -418,983 \cdot N$. Funkci lze definovat následujícím vztahem:

$$\sum_{i=1}^{N} -x_i \sin \sqrt{|x_i|} \tag{10}$$

3.2 Algoritmus rojení částic

Rojení částic nebo také anglicky Particle Swarm Optimization (PSO) [6, 7] je optimalizační technika, resp. algoritmus vytvořený na základě pozorování chování živočišných druhů v přírodě. Byla navržena a vytvořena R. Eberhartem a J. Kennedym [6]. Funguje na základě vytvoření nové populace částic a jejím postupným zkvalitňováním, následováním částice s nejlepší výslednou hodnotou účelové funkce, resp. vhodností [6].

3.2.1 Popis

Tento algoritmus podle literatury [6] probíhá cyklicky v iteracích. Na začátku jsou náhodně inicializovány pozice a rychlost částic. Po inicializaci dojde k výpočtu nové rychlosti podle vztahu 11 a na základě rychlosti k výpočtu pozic podle vztahu 12. Součástí výpočtů je i kontrola zda rychlost a pozice nepřesáhli své stanovené meze. Maximální rychlost je stanovena parametrem V_{max}. Následně jsou aktuální jednotlivé pozice předány účelové funkci jako vstupní argumenty. Po získání hodnoty účelové funkce, resp. vhodnosti je porovnána s nejlepší hodnotou aktuální částice. Pokud je získaná vhodnost menší, nastaví se tyto pozice a hodnota účelové funkce jako nejlepší dané částice. Dále je zjištěno, jestli není vypočtená hodnota účelové funkce menší než dosud nejlepší v celé populaci. Pokud je, přiřadí se tato aktuální hodnota a pozice jako nejlepší. Tento popis můžeme vidět zapsaný pseudokódem 1, který je inspirovaný literaturou [6].

Vztahy uvedené níže v pořadí slouží pro výpočet rychlosti, pozic a setrvačnosti [6]:

$$\begin{aligned} \vec{v}_d\,(t+1) = w \cdot \vec{v}_d\,(t) &+ c_1 \cdot rand \cdot \left(\vec{particleBest}_{i,d} - \vec{x}_{i,d}\,(t)\right) \\ &+ c_2 \cdot rand \cdot \left(\vec{globalBest}_d - \vec{x}_{i,d}\,(t)\right) \end{aligned} \tag{11}$$

kde,
w – setrvačnost (viz níže v 3.2.2), výsledek vztahu 13,
$\vec{v}_d\,(t+1)$ – rychlost částice následující iterace,
$\vec{x}_{i,d}\,(t+1)$ – pozice částice následující iterace,
$\vec{v}_d\,(t)$ – aktuální rychlost částice,
$\vec{x}_{i,d}\,(t)$ – aktuální pozice částice,
$rand$ – udává náhodné číslo v intervalu od nuly do jedné,
c_1, c_2 – učící faktory,

$\vec{particleBest}_{i,d}$ – nejlepší lokální pozice konkrétní částice,
$\vec{globalBest}_d$ – nejlepší pozice konkrétní částice v populaci.

$$\vec{x}_{i,d}(t+1) = \vec{x}_{i,d}(t) + \vec{v}_d(t+1) \tag{12}$$

kde,
$\vec{x}_{i,d}(t+1)$ – pozice částice následující iterace,
$\vec{x}_{i,d}(t)$ – aktuální pozice částice.
$\vec{v}_d(t+1)$ – rychlost částice následující iterace,

$$w = w_{start} - \frac{(w_{start} - w_{end}) \cdot iterace}{iterace_{max}} \tag{13}$$

kde,
w_{start} – počáteční hodnota setrvačnosti,
w_{end} – koncová hodnota setrvačnosti,
$iterace$ – aktuální iterace algoritmu,
$iterace_{max}$ – celkový počet iterací algoritmu (nebo také migračních kol [6]).

Algorithm 1 Pseudokód algoritmu rojení částic (Particle Swarm) inspirovaný literaturou [6]

```
for iterace = 0 to Migrace do
  for d = 0 to Počet částic do
    Vypočítej rychlost ze vztahu 11.
    Vypočti pozice ze vztahu 12.
    Cost = f_cost(pozice[d])
    if Cost < particleBestCost_d then
      particleBestCost_d = Cost
    end if
    if particleBestCost_d < globalBestCost then
      globalBestCost = particleBestCost_d
    end if
  end for
end for
```

3.2.2 Význam použitých parametrů PSO

V tomto algoritmu se vyskytují následující parametry [6]:

- **Dimenze** – je určena konkrétním problémem. Je to v podstatě počet argumentů účelové funkce.
- **Počet částic** – udává, kolik jedinců se bude v populaci vyskytovat.

- **Rozsah** – stanovuje rozsah, ve kterém se budou částice (jedinci) pohybovat.
- **Setrvačnost** – značí se (w), ovlivňuje rychlost částice, její hodnota je získána vypočtením podle vztahu 13. V počátku průběhu algoritmu ovlivňuje rychlost takovým způsobem, kdy částice prohledávají oblast po velkých skocích. U konce se pak pohybují v oblasti bližší nalezenému extrému.
- **Faktor** c_1 – tímto faktorem je upřednostňováno posunutí částice k aktuálně své nejlepší pozici.
- **Faktor** c_2 – tímto faktorem je upřednostňováno posunutí částice k nejlepší pozici v celé populaci.
- **Parametr** V_{max} – stanovuje maximální rychlost pohybu částice v prostoru.

3.3 Generátory pseudonáhodných čísel

Pro správné fungování vyžadují evoluční algoritmy generátory pseudonáhodných čísel, dále v textu a později zdrojovém kódu implementace budou označovány zkráceně už jen jako náhodná čísla. U těchto vygenerovaných hodnot je z důvodu lepší efektivity kladena podmínka, aby byly co nejvíce náhodné. Právě proto se v práci setkáme s následujícími generátory:

3.3.1 Mersenne Twister

Tento generátor, zkráceně označován jako MT19937 navrhli Makoto Matsumoto a Takuji Nishimura a má oproti ostatním generátorům několik výhodných vlastností navíc, jako je velmi velká perioda ($2^{19937} - 1$) a dobrá přesnost rozložení čísel [8]. Zahrnutí tohoto generátoru do implementace bude popsáno v kapitole 5.4 a 5.6.

3.3.2 Logistická rovnice

Logistická rovnice [9, 18] je nelineární rovnice publikovaná Robertem Mayem. Tato rovnice představuje určitý chaotický systém. Můžeme ji vidět definovanou ve vztahu 14. Rovnice se v čase průběžně vyvíjí.

$$x_{n+1} = r \cdot x_n \cdot (1 - x_n) \tag{14}$$

kde,
r – je parametr, jehož hodnota se pohybuje v intervalu $\langle 0, 4 \rangle$,
x – počáteční hodnota [18].

4 CUDA

CUDA (Compute Unified Device Architecture) [10] je paralelní architektura nasazená na grafických kartách společnosti NVIDIA. Byla vyvinuta a představená v roce 2006. Využívá se pro paralelní výpočet složitějších procesů na grafických kartách. Toto využití je daleko efektivnějším způsobem zpracovávání než u procesoru CPU. Procesy, které jsou nejčastěji zpracovávány, jsou např. vykreslování 3D obrazu, zpracování medií, kódování/dekódování videa, vykreslování obrazu či paralelní zpracování různých algoritmů. Pro vývoj na této platformě se využívá kromě jazyka C/C++ také FORTRAN, Java, Python nebo OpenACC. S touto platformou pracuje i paralelní MATLAB. [10]

4.1 Popis hardware

Jednotlivé grafické karty s podporou CUDA platformy můžeme rozdělit do tzv. anglicky nazývané „compute capability“ [10], v přímém českém překladu „výpočetní možnosti“. Tyto „compute capability“ mám určují konkrétní architekturu grafického zařízení. V podstatě udává jaké funkce či instrukce jsou na daném zařízení podporovány.

U této platformy docházelo v průběhu vývoje k různým hardwarovým změnám. Zařízení prvních „compute capability“ se skládají z multiprocesorů, které jsou založené na SIMD architektuře (viz podkapitola 4.1.1). Každý multiprocesor obsahuje paměť konstant, textur a sdílenou paměť. Paměť zařízení je pak přístupná všem multiprocesorům. [11]

Novější revize se skládají z vícevláknových multiprocesorů, přičemž jejich počet závisí na konkrétní implementaci v grafické kartě. Jednotlivé multiprocesory jsou navrženy tak, aby na nich mohlo běžet více bloků vláken současně a pokud jsou bloky ukončeny nahradí je další v pořadí. Platforma v těchto „compute capability“ je založena na architektuře SIMT. [10]

4.1.1 SIMD Architektura

V SIMD (Single Instruction, Multiple Data) architektuře zpracovávají multiprocesory stejné instrukce na základě jiných dat. Vlákna jsou rozdělována do skupin vláken tzv. warpů, přičemž každý warp má stejný počet vláken. Multiprocesor periodicky přepíná mezi warpy pokud jich zpracovává více. [11]

4.1.2 SIMT Architektura

Tato architektura (Single Instruction, Multiple Thread) je podobná architektuře SIMD, ale na rozdíl od ní umožňuje větvení kódu ve vláknech, tzn. že zpracování probíhá nezávisle na ostatních vláknech. [10]

Tak jako je tomu u SIMD architektury, principem je rozdělit vlákna postupně do warpů plánovačem warpů pro zpracování na multiprocesoru. Warp je u této architektury skupina 32 paralelních na sobě nezávislých vláken. Pokud je ve všech vláknech stejná instrukce je vykonána

paralelně. Pokud dojde k větvení způsobené na základě dat, jsou stejné větve ve vláknech zpracovány a ostatní vlákna jsou pozastavena. [10]

4.2 Popis programátorského modelu

Vývoj na této platformě vychází z toho, že existují dva systémy. Hostitel (CPU) a zařízení (GPU) [10, 11]. Průběh fungování si můžeme popsat na následujícím příkladu: jako první nakopírujeme data z paměti hostitele (RAM) do paměti zařízení (GPU) a poté spustíme naprogramované funkce, nazývané jako kernel, v každém z definovaného počtu vláken na zařízení paralelně. Zařízení zpracuje ve více vláknech data ze své paměti. Po zpracování překopírujeme zpět data z paměti zařízení do paměti hostitele (RAM). [11]

Jednotlivá vlákna můžeme organizovat do tzv. mřížek a bloků. Vlákna jsou rozdělena do bloků a jejich maximální počet v každém takovém bloku je omezen. Omezení je pak v každé „compute capability“ jiné, v „compute capability“ do 2.0 (včetně) je maximální počet 512. Od 3.0 do 7.0 je maximální velikost 1024 vláken na blok. Každá mřížka je pak složena z více bloků. Mřížky, bloky i vlákna mohou být organizovány až do maximálně tří dimenzí, tzn. že lze vytvořit např. matici jednotlivých vláken v bloku (např. při zpracování obrazu). [11]

Indexy jednotlivých vláken, bloků či mřížek je pak možné získat z následujících proměnných [10, 11], názvy jsou standardizovány přímo ve vývojovém prostředí CUDA platformy:

- `threadIdx` – tato proměnná obsahuje index aktuálního vlákna.
- `blockIdx` – určuje index aktuálního bloku.
- `blockDim` – udává velikost bloku, tj. počet vláken v bloku.
- `gridDim` – proměnná udávající velikost mřížky, tj. počet bloků v aktuální mřížce.

Proměnné `threadIdx` a `blockIdx` jsou datového typu `uint3`, který definuje tři složky pro tři hodnoty v jedné proměnné (např. `threadIdx.x`, `threadIdx.y`, `threadIdx.z`) a tyto hodnoty pak určují souřadnice daného vlákna v případě `threadIdx` či bloku `blockIdx`. [10, 11]

Na této platformě se vyskytují tři typy pamětí [11], do kterých mohou přistupovat všechny kernely. Paměti globální, konstant a textur. Dále se zde vyskytují registry, sdílené a lokální paměti, přičemž sdílené jsou přístupné pouze v rámci bloku a paměti lokální a registry pouze v rámci vlákna [11].

4.3 Knihovny a projekty na CUDA platformě

Existuje mnoho vytvořených a dostupných projektů, resp. implementací neuronových sítí s učením na CUDA platformě, vytvářených za účelem měření či dokazování určité hypotézy. Pak se také vyskytují jako „softwarové produkty“, jejímž cílem je pouze poskytnout konkrétní výsledek pro širokou oblast zaměření nebo jsou koncipovány jako učební příklady, jak by měla daná

implementace vypadat. Ne u všech, které se mi podařilo najít, je dostatečně zpracována dokumentace, proto jsou níže v této kapitole kromě pár vybraných implementací zahrnuty i některé vybrané nejznámější knihovny a frameworky spojené s CUDA platformou.

4.3.1 Knihovna cuDNN

Knihovna cuDNN [13] byla vytvořena za účelem zjednodušení a hlavně optimalizace při práci s hlubokými neuronovými sítěmi a jejím učením na paralelní platformě. Implementuje základní rutiny a poskytuje API rozhraní pro práci s nízko-úrovňovými a rutinními operacemi na datech. Mezi hlavní cíle patří efektivní práce s pamětí GPU. Tato knihovna také poskytuje implementace aktivačních funkcí, jako je Sigmoid, hyperbolický tangens nebo Softmax.

4.3.2 Caffe

Tento framework byl vytvořen Yangqing Jia pro učení hlubokých neuronových sítí na UC Berkley a je více popsán v článku [12]. V následujícím odstavci se nachází jeho zestručněný popis.

Architektura je založena na modularitě a navržena tak, aby modely a hluboké neuronové sítě byly nadefinovány po vrstvách v obyčejném textu namísto přímo v kódu. Využití nachází kvůli své rychlosti ve výzkumných projektech a v průmyslových aplikacích. [12]

4.3.3 TensorFlow

TensorFlow [14] je open source knihovna nebo také platforma, která se zaměřuje především na strojové učení např. hlubokých neuronových sítí. Tato knihovna (nebo také platforma) pracuje na více zařízeních jako je např. GPU, vícejádrové procesory (CPU) nebo také na modulech TPU. Využití nachází v experimentech ve výzkumu nebo v oblasti produkce software. Více o této platformě pojednává článek [14].

4.3.4 Neuronová síť s učením Backpropagation na CUDA platformě

Následující článek [15] pojednává o učení neuronové sítě pomocí metody Backpropagation na CUDA platformě a srovnává čas učení na CPU s učením na GPU.

4.3.5 Implementace neuronové sítě s využitím CUDA a OpenMP

Další implementací s využitím CUDA architektury je projekt popsaný v článku [16], srovnává doby běhu implementací neuronové sítě, na které je založen detekční systém textu na CPU, CUDA GPU s OpenMP a bez něj.

4.3.6 Hluboká konvoluční neuronová síť a ImageNet

V příspěvku [17] je popsáno učení hluboké konvoluční [4] neuronové sítě na obrázcích s vysokým rozlišení ImageNet a jejich klasifikace s použitím GPU.

[4] Konvoluční neuronové sítě se řadí mezi speciální třídu hlubokých neuronových sítí [17, 13].

5 Návrh a implementace

V této části si popíšeme implementace vícevrstvé perceptronové neuronové sítě s dopředným šířením a algoritmu rojení částic sekvenčně v jazyce C++ a následně paralelně pro CUDA platformu.

Celé řešení bakalářské práce bylo vyvíjeno a odladěno v prostředí Visual Studio 2015 Enterprise pod operačním systémem Windows 10 s použitím nástrojů NVIDIA Nsight Visual Studio Edition ve verzi 6.0 a CUDA Toolkit 10.0 na grafické kartě s „compute capability“ 6.1. Práce je rozdělena do dvou hlavních projektů. Projekt sekvenční části je pojmenován *bp_serial_cho0147* a paralelní části *bp_parallel_cuda_cho0147*.

V „solution“ se nachází i jeden vedlejší projekt pojmenovaný *SinusDataSetGenerator*, který slouží k vytvoření nových trénovacích a testovacích dat. Nevyžaduje žádný uživatelský přístup. Jeho podrobnější popis bude více rozveden v následující kapitole 6.

Zdrojové kódy se nachází v celkem 72 souborech, které z větší části obsahují jen třídy, tedy celkem dohromady 20 různých tříd a 4 druhy výjimek. Zbytkem jsou soubory s funkcemi nacházející se především v projektu s paralelní implementací. V souborech *SerialApplication.h*, *ParallelApplication.h* a *Parameters.h* se nacházejí mimotřídní konstantní proměnné. Některé třídy a soubory mezi projekty jsou duplicitní, protože byly potřeba v obou projektech.

Programátorská dokumentace [5] pro každý ze všech tří projektů je napsána kompletně v českém jazyce podle zadání práce, které je také v češtině. Nachází se ve formátu HTML a XML v elektronické příloze. V dokumentaci nenalezneme popisy triviálních metod, jako jsou metody vracející hodnoty privátních instančních proměnných.

V sekvenční implementaci algoritmu rojení částic, resp. třídě `SerialPSO`), byla pro potřeby ladění při vývoji, vytvořena funkce sbírající informace z průběhu algoritmu. Sbírána jsou data o nejlepší částici v populaci. Pro výstup byl vybrán typ souboru CSV. Metody a proměnné spojené s touto funkcí jsou ve zdrojovém kódu v elektronické příloze zakomentovány a není použita ani při testování. Více si popíšeme v podkapitole 5.3.4.

V následující popisné kapitole budou pro lepší orientaci třídy sekvenční i paralelní implementace `SerialApplication`, `ParallelApplication`, `ParametersReader`, `Parameters` a `CsvFile` označovány jako uživatelské rozhraní.

Vše z výše uvedeného včetně výsledných binárních souborů aplikací můžeme najít v elektronické příloze.

5.1 Uživatelské rozhraní, vstupy a výstupy obou aplikací do souboru

Uživatelské rozhraní je koncipováno jako konzolové uživatelské prostředí. Obě aplikace mají funkcionálně stejné rozhraní, jen ve zdrojovém kódu jsou z důvodu rozdílné implementace neuronové sítě a jejího učení odlišnosti. Největší touto odlišností je kontrola grafické karty na pod-

[5] Je vygenerována pomocí dokumentačního nástroje Doxygen distribuovaného pod licencí GNU General Public License a je dostupný na webové adrese: `http://www.doxygen.nl/index.html`.

poru CUDA platformy, nastavení a výpis informací spojené s grafickou kartou. Od spuštění až po ukončení nevyžaduje toto rozhraní žádnou interakci ze strany uživatele.

Vstup je řešen načtením parametrů z textového souboru. Výchozí název a cesta je nastavena přímo v kódu na hodnotu *parametry_nastaveni.txt* v souborech *bp_cho0147.cpp* a *bp_cuda_cho0147.cpp* v proměnné `parametersFilePath`. Pokud by byla potřeba číst z jiného souboru, je možnost spustit aplikace přes příkazovou řádku s parametrem, kde je tímto parametrem cesta s názvem souboru.

Výstupem je výpis souhrnných a jen hlavních informací do konzole. Podrobnější výstup je zapisován do souboru, jehož název se skládá z textového řetězce v konstantní proměnné `DETAILED_RESULTS_NN_PATH` nebo `DETAILED_RESULTS_PSO_PATH` v souboru *Parameters.h* a z přetypované hodnoty na datový typ `integer` získané z funkce `time` knihovny *time.h* jazyka C++. Vstupní argument je nastaven hodnotu `NULL`. Výběr proměnné, ze které je řetězec získán, závisí na funkci aplikace.

Statické třídy `SerialApplication` a `ParallelApplication` obsahují metody pro výpis informací do konzole a souboru.

5.1.1 Pomocné třídy pro čtení parametrů ze souboru

Pro čtení parametrů jsou vytvořeny třídy `ParametersReader` a `Parameters`. Třída `ParametersReader` má metody pro čtení a ukládání hodnot parametrů ze souboru do dynamického pole datového typu `string`. Tato „čtečka parametrů“ je navržena pro univerzální použití. Jednotlivé parametry jsou definovány vstupními argumenty v konstruktoru. Třída `Parameters` rozšiřuje třídu `ParametersReader` a zároveň má definovány parametry přímo v kódu.

Hodnoty parametrů získané ze souboru jsou ošetřeny na špatný formát hodnot a případné záporné nebo nulové hodnoty. Podrobněji jsou jednotlivé parametry rozpracovány spolu s popisem v příloze A.

Tyto třídy využívají kromě výjimek nacházejících se v knihovnách jazyka C++ také vlastní výjimky, kterými jsou a které znamenají:

- `FileNotFound` – nenalezený soubor.
- `ParameterNotFound` – nebyl nalezen parametr.
- `BadNumberOfValues` – špatný počet hodnot ve vícehodnotovém parametru.
- `BadFormatOfValue` – špatný formát hodnoty v parametru.

5.1.2 Trénovací, testovací sada vzorů a třída CsvFile

Pro trénovací a testovací sadu dat byl vybrán jednoduchý formát CSV. Máme zde možnost definovat oddělovač hodnot v souboru s parametry. Více o parametrech najdeme v příloze A.

Obě sady obsahují jak vstupy, tak očekávané výstupy. Od prvního sloupce do počtu vstupů se nachází hodnoty vstupní. Od sloupce následujícího do konce řádku pak hodnoty výstupní. Organizace těchto vstupů a výstupů byla již popsána v podkapitole 2.3.1 a je inspirována literaturou [1]. Všechny hodnoty jsou v těchto souborech předem normalizovány metodou MIN-MAX [21] do intervalu konkrétní používané aktivační funkce. Tuto metodu si popíšeme později v podkapitole 6.2 v kapitole testování.

Třída CsvFile slouží ke čtení a zápisu z/do CSV souboru. Hodnoty získané ze souboru jsou uloženy v dvourozměrném dynamickém poli `float` hodnot. Třída obsahuje také metodu pro přidávání nového řádku `float` hodnot pro výpis do CSV souboru.

5.2 Třídy AbstractNeuralNet a AbstractPSO

Tyto abstraktní třídy jsou vytvořeny z důvodu použití stejných proměnných a metod v obou implementacích. Abstraktní třída `AbstractNeuralNet` má v sobě definovanou metodu `calculateCountWeights`, která vypočítá počet vah v síti. Přehled jednotlivých atributů a metod těchto tříd můžeme vidět v třídním diagramu v příloze B.

5.3 Sekvenční implementace

V této části si popíšeme objektový návrh tříd tvořící neuronovou síť `SerialNeuralNet`, `Neuron`, optimalizační algoritmus PSO rozdělený mezi třídy `SerialPSO` a `Particle` a generátory náhodných čísel. Vztahy mezi třídami si postupně rozebereme v jednotlivých podkapitolách a najdeme je také ve třídním diagramu v příloze B. Stanovení velikosti dimenze u algoritmu rojení částic podle počtu vah sítě a nastavení pozic jako vah sítě je inspirováno článkem [5].

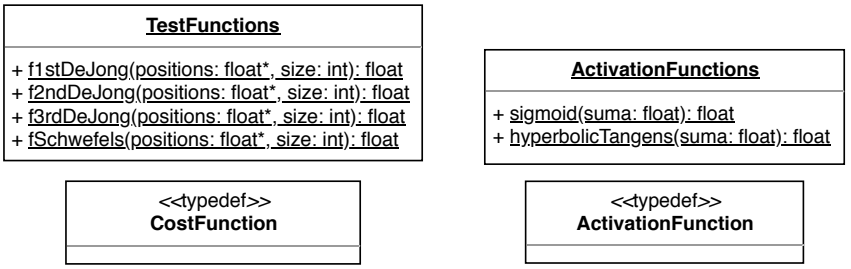
Ve výše zmíněných třídách nejsou ošetřovány žádné vstupní hodnoty z důvodu možného zpomalování, s ohledem na to, že bude měřena doba vykonávání konkrétních částí tohoto zdrojového kódu. Hodnoty přebírané z uživatelského vstupu jsou kontrolovány již ve třídě `Parameters` a `CsvFile`.

5.3.1 Statické třídy TestFunctions a ActivationFunctions

Tyto třídy obsahují implementace vybraných funkcí. U `TestFunction` jsou to testovací účelové funkce [6], které jsme již prošli v teoretické části v kapitole 3.1.4 a budou použity k otestování obou implementací algoritmu PSO. Ve třídě `ActivationFunctions` se pak nacházejí funkce probrané v kapitole 2.3.2.

V hlavičkových souborech `TestFunctions.h` a `ActivationFunctions.h` se nacházejí typové deklarace ukazatelů na funkce, které jsou také ve stejnojmenných třídách. Ukazatele na tyto funkce jsou pak využity pro předávání do metod jako je v případě `TestFunctions` metoda `run` ve třídě `SerialPSO` a pro `ActivationFunctions` konstruktor třídy `SerialNeuralNet` a metoda `calculateOutput` třídy `Neuron`. V souboru `ActivationFunctions.h` se nachází jedna mimotřídní konstantní proměnná pojmenovaná `LAMBDA` nastavená na hodnotu 1 a to z důvodu po-

psaného v poznámce pod čarou v podkapitole 2.3.2.1, že nedojde k ovlivnění výsledku funkce. Třídy i s ukazateli můžeme vidět na obrázku 5.

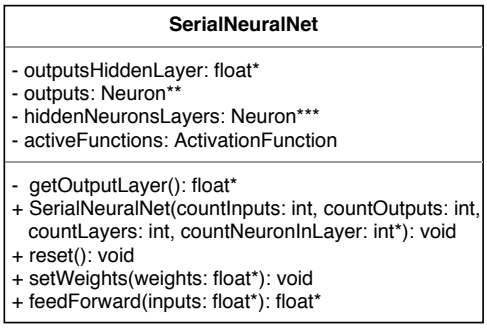


(a) Třída TestFunctions (b) Třída ActivationFunctions

Obrázek 5: Výřez tříd TestFunctions a ActivationFunctions z třídního diagramu.

5.3.2 Třída SerialNeuralNet

Třída `SerialNeuralNet`, jak už z názvu vyplývá, reprezentuje vzor pro sekvenčně implementovanou neuronovou síť a dědí z abstraktní třídy `AbstractNeuralNet`. Přehled metod a proměnných třídy `SerialNeuralNet` můžeme vidět na obrázku 6. Implementovaná vícevrstvá neuronová síť je navržena univerzálně k řešení různých problémů. Je možné navolit pomocí parametrů počet vstupů, výstupů, vrstev a jednotlivých neuronů v těchto vrstvách.



Obrázek 6: Třída SerialNeuralNet.

Vstupní vrstva je tvořena jednorozměrným polem, obsahující `float` hodnoty. Toto pole není v instanci třídy nijak dlouhodobě udržováno. Předává se jen jako parametr metodě `feedForward` a není také vůbec upravováno. Předávané pole je jedním řádkem konkrétního CSV souboru a vstupní hodnoty jsou vybrány na základě počtu vstupů v proměnné `countInputs` od nultého indexu po `countInputs - 1`.

Jak již bylo zmíněno v podkapitole 5.1.2 očekává se, že vstupní hodnoty předem normalizujeme pro konkrétní aktivační funkci, právě z důvodu této univerzálnosti. Skrytá vrstva, resp. skryté vrstvy jsou pole ukazatelů odkazující na další pole referencí na instance třídy `Neuron`. Výstupní vrstva je pak pole ukazatelů na instance zmíněné třídy. Váhy mezi vrstvami (předchozí a následující) se vždy nacházejí v jednotlivých instancích třídy `Neuron` následující vrstvy. Toto

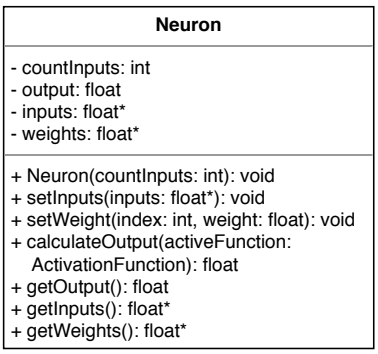
rozvržení bylo implementováno podle popisu fungování vícevrstvých sítí již popsáných v podkapitole 2.3.1 z literatury [1, 2].

Popis nejrozsáhlejších metod třídy:

- `feedForward` – přijímá pole vstupů přes vstupní argument metody. Nastaví vstupy jednotlivým neuronům, resp. instancím třídy `Neuron` a volá postupně na každou instanci metodu `calculateOutput` třídy `Neuron`. Zároveň jí předává ukazatel na aktivační funkci. Jakmile dokončí přepočet jedné vrstvy, vloží přepočtené výstupy aktuální vrstvy do dočasného pole a předá je pomocí metody `setInputs` třídy `Neuron` jako vstupy objektům v další vrstvě. Pak se průběh cyklicky opakuje podle počtu skrytých vrstev. Celý tento proces je dopředné šíření ve vícevrstvé síti popsané v podkapitole 2.3.1 a je inspirováno z popisu dopředného šíření literaturami [1, 2].
- `setWeights` – nastavuje váhy sítě. Jako vstupní parametr přijímá jedno pole reálných hodnot. Ty jsou následně postupně nastavovány instancím třídy `Neuron` od první v první skryté vrstvě až po poslední ve výstupní vrstvě.
- `reset` - nastaví a upraví všechny privátní instanční proměnné tak, jak byly při jejím vytvoření v konstruktoru.

5.3.3 Třída Neuron

Třída `Neuron` reprezentuje vzor pro umělý neuron, který byl popsán již v první kapitole 2.1 podle literatury [1, 2]. Konkrétní instance obsahuje ukazatel na pole vstupů a vah datového typu `float`, hodnotu vypočteného výstupu, která je ve výchozím stavu při vytvoření nastavena na hodnotu `NULL`. Jednotlivé proměnné a metody třídy `Neuron` můžeme vidět na obrázku 7.

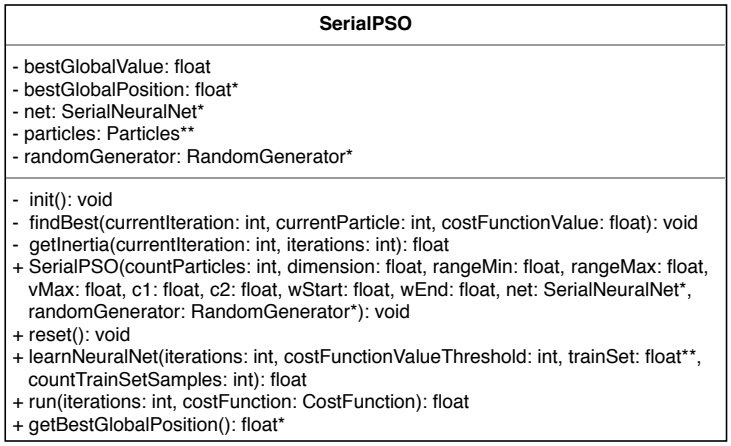


Obrázek 7: Třída Neuron.

Nově vypočtený výstup z neuronu získáme zavoláním metody `calculateOutput`. Tato metoda má jako vstupní argument ukazatel na aktivační funkci ze statické třídy `ActivationFunctions`. Metoda vypočte sumu vah a vstupů podle vztahu 1 [1] a zavolá tuto aktivační funkci, které předá sumu. Výslednou hodnotu si uloží v neuronu do proměnné `output` a vrací ji zároveň jako návratovou hodnotu.

5.3.4 Třída SerialPSO

Tato třída má implementovány metody simulující průběh algoritmu PSO přizpůsobeném pro učení neuronové sítě. Neuronová síť, resp. instance třídy `SerialNeuralNet` je předána ukazatelem do této třídy v konstruktoru. Přehled jednotlivých metod a proměnných můžeme vidět na obrázku 8.



Obrázek 8: Třída SerialPSO.

Pro lepší přehlednost o průběhu vykonávání algoritmu je do sekvenční implementace zahrnuta možnost sběru informací o nejlepší částici v populaci pro každou iteraci. Tato možnost je ale z důvodu zpomalování neaktivní a sloužila pouze během vývoje. Pokud by jsme potřebovali využívat této možnosti, musíme odkomentovat a zakomentovat řádky kódu a znovu zkompilovat. Seznam těchto jednotlivých řádků k odkomentování/zakomentování:

- *bp_serial_cho0147.cpp*

  odkomentovat – 19, 20, 41 – 43, 64, 113 – 117, 122, 141 – 143, 152, 168 – 172, 183 – 185, 194.

  zakomentovat – 18, 118, 119, 123, 153, 173, 174.

- *AbstractPSO.h*

  odkomentovat – 30.

- *SerialPSO.h*

  odkomentovat – 34, 36, 37.

  zakomentovat – 38.

- *SerialPSO.cpp*

  odkomentovat – 54, 69 – 85, 102, 103, 115.

  zakomentovat – 101.

Pro sběr těchto informací je do třídy zahrnuta metoda `createPSOBestParticleRow`, která přidává řádky. Řádkem je dynamické pole `float` hodnot v datovém kontejneru `vector` instance třídy `CsvFile`. Ukazatel na ní najdeme ve třídě `SerialPSO` v proměnné `psoBestParticles`. Zakomentované metody a proměnné se nevyskytují v třídním diagramu v příloze B ani na obrázcích tříd.

Syntakticky nejrozsáhlejší metody této třídy jsou:

- `learnNeuralNet` – spustí algoritmus PSO podle pseudokódu 1 [6] s úpravou pro učení neuronové sítě. Po spuštění volá metody `init` a pak v cyklu `getInertia`. V cyklu dále pokračuje. Místo účelové funkce je zde použit vztah 5 [1] pro výpočet chyby. Před výpočtem chyby jsou nastaveny instanci třídy `SerialNeuralNet` nejlepší pozice v populaci `bestGlobalPosition` její metodou `setWeights` jako váhy. Následně je volána její další metoda `feedForward` a přes vstupní argument této metody je předáno pole `float` hodnot z dvourozměrného pole nacházející se v ukazateli `trainSet`. Návratovou hodnotou je pole aktuálních výstupů (proměnná `currentOutputs`), ty jsou pak spolu s očekávanými výstupy přepočítány. Vypočtená chyba (`costFunctionValue`) je předána metodě `findBest`. Celý tento průběh se cyklicky opakuje podle počtu iterací. Je ale možnost ukončit cyklus předčasně pokud chyba sítě klesne pod stanovenou hodnotu v konstantní proměnné `NEURAL_NETWORK_ERROR_EXIT_THRESHOLD` v souboru *Parameters.h*. Výchozí hodnota je nastavena na -1. K předčasnému ukončení tedy nikdy nedojde. V případě změny hodnoty by musel být projekt znovu zkompilován.

- `run` – spustí algoritmus PSO podle pseudokódu 1 [6]. Na začátku volá metody `init` a pak v cyklu `getInertia`. V tom samém cyklu pak také volá účelovou funkcí, kterou je testovací funkce statické třídy `TestFunction`, předaná ukazatelem přes vstupní argument metody. Nakonec volá metodu `findBest`. Zde popsaný postup se opakuje podle počtu iterací. Tato metoda slouží pouze k otestování funkčnosti algoritmu PSO.

- `reset` – nastaví hodnoty proměnné `bestGlobalValue` do stavu, v jakém byla při vytvoření konkrétní instance této třídy. Nastaví ji tedy na nejvyšší hodnotu datového typu `float`. Dealokuje a opět alokuje dynamická pole `bestGlobalPosition` a `particles`.

- `init` – privátní metoda inicializuje algoritmus PSO, tedy vytvoří instance třídy `Particle` a volá jejich veřejné metody `setRandomPosition`, `setBestPosition` a `setRandomVelocity`, kterými nastaví náhodné pozice, nejlepší pozice a rychlost. Do `bestGlobalPosition` přiřadí aktuální pozice první instance třídy `Particle` v poli `particles`. Je volána metodami `learnNeuralNet` a `run`.

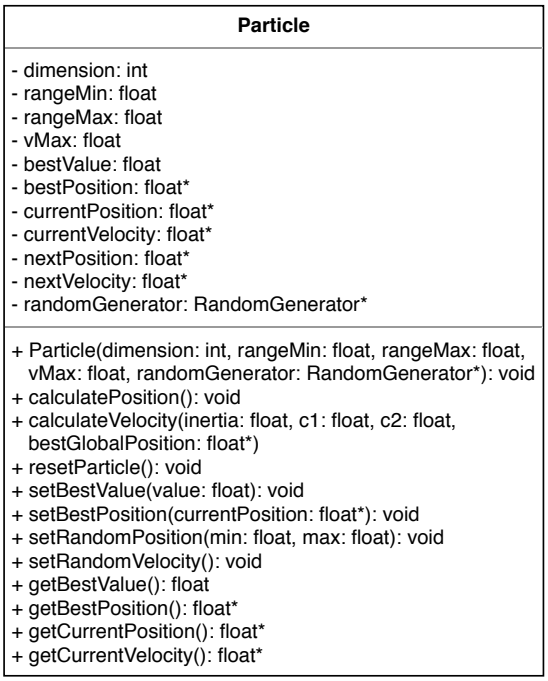
- `findBest` – nastavuje případnou nejlepší hodnotu účelové funkce a také nejlepší pozice v proměnné `bestPosition` a poli `bestGlobalPosition`.

- `getInertia` – vypočítává setrvačnost podle vztahu 13 z [6].

Hodnoty dynamických polí pozic a rychlostí jsou mezi sebou kopírovány pomocí generické funkce `copyArray`, jejíž definici by jsme našli v souboru *stdafx.h*. Vyskytuje se v tomto souboru, protože je v celém projektu tato pomocná funkce jediná. Využití nachází funkce i ve třídě `Particle`.

5.3.5 Třída Particle

Instance této třídy reprezentuje částici v populaci PSO a obsahuje pole pozic, rychlosti. Přehled metod a proměnných uvidíme na obrázku 9.



Obrázek 9: Třída Particle.

Metoda `reset` nastaví v instanci všechny privátní instanční proměnné a pole do výchozího stavu, tj. vynulování jednotlivých polí konkrétní instance třídy `Particle` a nastavení náhodných aktuálních pozic a rychlostí metodami `setRandomPosition` a `setRandomVelocity`. Tato instance pak dále pokračuje v průběhu algoritmu. Metoda je volána metodami `calculatePosition` a `calculateVelocity` jen pokud nastane případ, že některá z nově vypočtených pozic překročila povolený rozsah (od `rangeMin` po `rangeMax`) nebo pokud při výpočtu rychlosti překročí maximum stanovené v proměnné `vMax`. Nová rychlost nebo pozice je tedy ihned po výpočtu jedné hodnoty v celém vektoru testována a pokud překročí meze je uvedena do výchozího stavu metodou `reset`. Výpočet pak u `calculatePosition` a `calculateVelocity` probíhá podle vztahů z literatury [6], kde pro `calculatePosition` platí vztah 12 a pro `calculateVelocity` vztah 11.

5.4 Generátory náhodných čísel

Rozhraní `RandomGenerator` je implementováno třídami `MersenneTwister19937` a `SeriaLogisticMap`, můžeme je vidět na obrázku 10. Konkrétní instance jedné z těchto tříd je na základě parametrů získaných ze souboru vytvořena metodami statické třídy `SerialApplication` a předána prostřednictvím tohoto rozhraní do instance třídy `SerialPSO`. Tzv. „seed“ je pro generátor MT19937 [19] nastaven na hodnotu získanou přetypováním návratové hodnoty funkce `time` C++ knihovny *time.h* se vstupním argumentem `NULL` na datový typ `unsigned int` a vynásoben náhodnou hodnotou získanou z funkce `rand` C++ knihovny *random*. V případě logistické rovnice jsou vynásobeny mezi sebou dvě náhodné hodnoty C++ knihovny *random* a upraveny do intervalu $(0, 1)$. Nastavení „seed“ hodnoty metodou `createRandomGenerator` ze statické třídy `SerialApplication` můžeme vidět v následující ukázce ze zdrojového kódu 1.

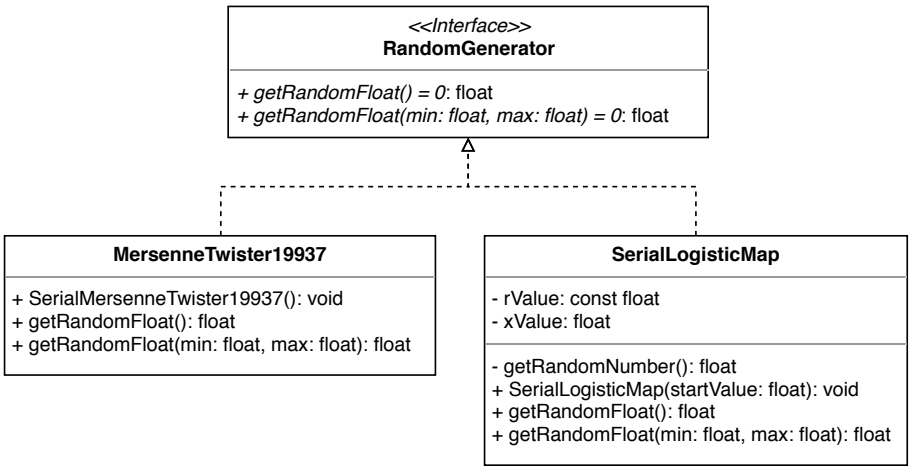
```
RandomGenerator* SerialApplication::createRandomGenerator(string generatorName)
{
   transform(generatorName.begin(), generatorName.end(), generatorName.begin(),
        tolower);
   srand((unsigned int)time(NULL));

   if (generatorName == "mt19937" || generatorName == "mersenne_twister")
      return new MersenneTwister19937((unsigned int)rand() *
         (unsigned int)time(NULL));
   else if (generatorName == "logisticka_rovnice" ||
            generatorName == "logistic_map")
      return new SerialLogisticMap((((float)rand()) / (float)RAND_MAX) * (((
          float)rand()) / (float)RAND_MAX));
   else
      return new MersenneTwister19937((unsigned int)rand() *
         (unsigned int)time(NULL));
}
```

Výpis 1: Metoda SerialApplication::createRandomGenerator

Třída `MersenneTwister19937` je využita pouze jako „obalovací“ třída 32. bitového generátoru MT19937 [19] z knihovny *random*. Náhodná čísla jsou pak generována pomocí „class template“ `uniform_real_distribution` ze stejné knihovny.

Třída `SerialLogisticMap` byla vytvořena za účelem generování čísel podle logistické rovnice vztahu 14 z [9, 18]. Implementována je v metodě `getRandomNumber` a náhodná hodnota je uložena v privátní instanční proměnné `xValue`. Zbývající metody upravují hodnotu do stanoveného rozsahu. Metoda `getRandomFloat` bez vstupních argumentů přetěžuje metodu `getRandomFloat` se dvěma argumenty. Nastavuje argumenty `min` a `max` na maximální a minimální hodnotu da-

tového typu `float` pomocí „class template“ `numeric_limits` C++ knihovny *limits*. Konstanta `rValue` je nastavena přímo ve zdrojovém kódu na hodnotu $3,9999$ na základě doporučení z diplomové práce [18].



Obrázek 10: Rozhraní RandomGenerator a třídy MersenneTwister19937 a SerialLogisticMap.

5.5 Paralelní CUDA implementace

V této podkapitole si především popíšeme paralelní návrh rozložení neuronové sítě a algoritmu PSO, včetně generátorů náhodných čísel na CUDA platformě. Tato implementace vychází ze sekvenční implementace, kterou jsme si popsali v podkapitole 5.3, ale na rozdíl od sekvenční části je jen částečně objektová a navržena pouze pro paralelní platformu CUDA. V této podkapitole si více popíšeme samotnou paralelizaci než postupný průběh jednotlivých metod, který je podobný sekvenční implementaci a byl popsán již tam. Stejně také jako v případě sekvenční implementace je stanovení dimenze algoritmu PSO podle počtu vah sítě a nastavení pozic jako vah sítě inspirováno článkem [5].

Instance tříd mají uloženy v privátních instančních proměnných ukazatele do paměti grafické karty a obsahují metody vytvořené pro hostitelský systém, tj. zpracování probíhá na CPU. Kód kernelů a funkcí na GPU zařízení mají deklarace a definice zapsány v hlavičkových a zdrojových souborech tříd, které je využívají. V této části se dále setkáme s třídami `ParallelNeuralNet`, `ParallelPSO`, `ParallelLogisticMap`, `MersenneTwister19937` a rozhraním `ParallelRandomGenerator`. Třídní diagram můžeme vidět v příloze C.

5.5.1 Hlavní a pomocné funkce

Nalezneme je v souboru *mainFunctionLib.h*. Je to jen „knihovna“ šesti `inline` funkcí. První dvě jsou typu `bool`, druhé dvě typu `void` a návratovým typem posledních je `integer`, podrobněji si jejich přesnější účel vysvětlíme v podkapitole 5.5.3. V seznamu níže jsou stručně popsány všechny funkce v souboru:

- `checkSupportingCUDA` – kontroluje zda jsou na počítači, kde bude aplikace běžet, zařízení podporující CUDA platformu.
- `connectCUDADevice` – nastaví vybrané zařízení s podporou CUDA podle vstupního argumentu `deviceNum` této funkce.
- `checkErrorCuda` a `checkErrorCuRAND` – kontrolují volané funkce vracející návratový typ `cudaError_t` na chybu. V případě že je chyba nalezena, vypíše jí do konzole a ukončí aplikaci. Touto funkcí jsou ošetřeny všechny funkce vracející návratový typ `cudaError_t`.
- `getCountPart` – nalezne a vrátí počet částí, na které bude rozděleno zpracovávané pole. Kromě toho také vypočte a nastaví argumentu `countInLastPart` počet prvků či vláken v poslední části.
- `getCurrentCount` – na základě vstupních argumentů nalezne a vrátí počet prvků či vláken v aktuální části rozděleného a zpracovávaného pole.

5.5.2 Rutinní funkce pro správu pole ukazatelů

Definice těchto rutinních funkcí pro práci s poli ukazatelů se nachází v hlavičkovém souboru *arraysOfPointersFunctionLib.h*. Všechny funkce alokují pole ukazatelů v paměti GPU a jsou zpracovávány pod CPU (hostem), jedná se o `inline void` funkce. Aby nedocházelo ke zpomalování nejsou vstupní argumenty nijak ošetřeny. V následujících odrážkách jsou všechny stručně popsány:

- `createArrayOfPointers` – alokuje dvourozměrné pole `float` hodnot. Výchozí hodnotou vstupního argumentu `inputArray` je hodnota `nullptr`, pokud je ale předán ukazatel na dvourozměrné pole v paměti RAM, nakopírují se z něj hodnoty do nově alokovaného pole.
- `createArrayOfPointersToArrayOfPointers` – alokuje pole ukazatelů obsahující další pole s ukazateli na pole `float` hodnot. Nadefinovány s tímto názvem jsou dvě funkce. Liší se v typu posledního argumentu `inputArray`. První je typu `float**` a druhý `float***`. Ve druhém případě jsou pole stejná. Ale v prvním případě je vícerozměrné pole cyklicky kopírováno do pole ukazatelů v paměti GPU podle hodnoty ve vstupním argumentu `newArraySizeX`. Můžeme vložit i hodnotu `nullptr`, je ale nutné uvést i typ. Např.: `(float**)nullptr` nebo `(float***)nullptr`.
- `copyArrayOfPointersToHost` – zkopíruje obsah předaného zdrojového pole v paměti grafické karty do cílového v paměti RAM.
- `deleteArrayOfPointers` – dealokuje matici vytvořené funkcí `createArrayOfPointers`.
- `deleteArrayOfPointersToArrayOfPointers` – dealokuje pole ukazatelů v paměti GPU.

5.5.3 Návrh rozdělení do více vláken

Algoritmus PSO a vícevrstvá neuronová síť používají v této paralelní části práce velké množství polí hodnot a ukazatelů. Ukazatele odkazují na další pole hodnot nebo ukazatelů. Několika násobné reference zanořené v jednotlivých polích se vyskytují ve třídách `ParallelNeuralNet` a `ParallelPSO`. Velikost jednotlivých polí pak závisí na jejich konkrétním účelu. Buď je to velikost dimenze (počtu vah neuronové sítě) nebo počet částic.

V návrhu jsou kombinovány celkem tři druhy možných rozdělení do vláken. První z nich se týká stejných operací, vykonávaných za sebou nad jednotlivými prvky konkrétního pole (především ve třídě `SerialPSO`), které jsou navzájem na sobě nezávislé. Bylo vhodné ve třídě `ParallelPSO` zvolit vykonávání těchto operací nad každým prvkem současně ve vláknech. CUDA umožňuje také rozdělovat i do bloků vláken [10], což souvisí z dalším druhem. Jednotlivé částice populace PSO podle pseudokódu 1 z [6] jsou také v některých úsecích vykonávání na sobě navzájem nezávislé a proto je lze také paralelizovat rozdělením částic na bloky, kde jejich vlákna budou zpracovávat pole těchto částic. Posledním druhem je rozdělení jednotlivých částic na vlákna jednoho bloku v případech, kdy bylo horší paralelizovat konkrétní operace nad částicemi.

5.5.4 Řešení omezení počtu vláken v jednom bloku

Jak již bylo zmíněno v podkapitole 4.2, počet vláken v bloku je omezen [10]. Zde je popsáno možné a obecně jednoduché řešení tohoto problému.

Výchozí nastavená hodnota v implementaci je zapsána ve statické proměnné `COUNT_MAX_THREADS_PER_BLOCK` ve statické třídě `ParallelApplication` a inicializována na hodnotu 1024 [6]. Ale během volání statické metody `printGPUInformation` stejné třídy je nastavena na hodnotu získanou z `cudaGetDeviceProperties` [10] nacházející se v CUDA knihovně. Pokud by tedy byla velikost zpracovávaného pole větší jak je tato omezující hodnota, kernel by skončil chybou.

V případě, že nastane možnost, kdy je pole větší jak hodnota omezení, je rozděleno toto pole ke zpracování do několika částí. Prvním částem rozděleného pole je nastavována vždy největší délka až do hodnoty omezení. U poslední části je pak nastaven zbytek délky. Postupné vykonávání na jednotlivých částech pole je serializováno do cyklu. Ukázku můžeme vidět v následujícím upraveném zdrojovém kódu 2. Ve zdrojovém kódu implementace nacházející se v elektronické příloze najdeme tento zdrojový kód pouze s konkrétním kernelem, nastaveným počtem bloků (jeden blok nebo počet částic) a nastavenými argumenty, přičemž proměnná `currentThreadsStartIndex` je předávána vždy. Metoda `getCountPartThreads` vrací počet částí pole do kterých bude pole rozděleno a zároveň ukládá do proměnné `countThreadsInLastPart` zbývající počet vláken v poslední části pole. Metoda `getCurrentCountBlock` vrací počet vláken pro aktuálně zpracovávanou část pole. Proměnná `currentThreadsStartIndex` má v sobě uložen

[6] Stanovena podle grafické karty NVIDIA GeForce GTX 1060 6 GB, na které byla implementace vyvíjena.

index na pozici v poli od které se začne v dalších zpracovávaných částech. Ten je předán jako jeden z argumentů do kernelu.

```
int currentThreadsStartIndex = 0;
int currentCountThreads = 0;
int countThreadsInLastPart = 0;
int countPartThreads = getCountPartThreads(arraySize, &
    countThreadsInLastPart, this->countMaxThreadsPerBlock);

for (int j = 0; j <= countPartThreads; j++)
{
   currentCountThreads = getCurrentCountThreads(arraySize, &
      countThreadsInLastPart, j, countPartThreads, this->
      countMaxThreadsPerBlock);

   kernel<<< countBlocks, currentCountThreads >>>(currentThreadsStartIndex)

   currentThreadsStartIndex += currentCountThreads;
}
```

Výpis 2: Abstraktní kód serializace paralelního zpracování částí pole

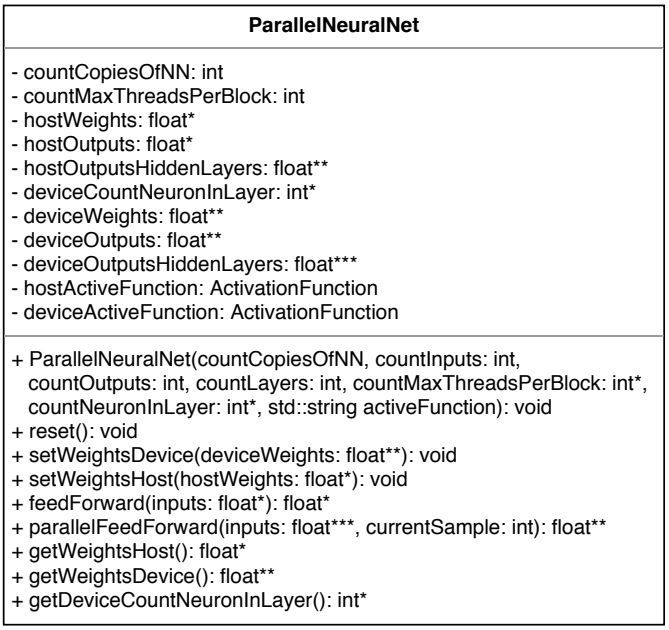
5.5.5 Třída ParallelNeuralNet

Instance této třídy obsahuje několik kopií neuronových sítí, aby bylo možno paralelně zpracovávat částice v instanci třídy `ParallelPSO`. Tento počet je definován vstupním argumentem konstruktoru třídy. Konstruktoru je předán počet částic algoritmu PSO.

Neuronové sítě jsou složeny ze tří ukazatelů do paměti GPU. Privátní instanční proměnné `deviceOutputs` a `deviceWeights` obsahují pole ukazatelů na pole `float` hodnot, které představují v případě první proměnné výstupy výstupních vrstev a u druhé váhy jednotlivých neuronových sítí. Proměnná `deviceWeights` je inicializována až při volání metody `setWeightsDevice` instancí třídy `ParallelPSO`, která jí předá jen ukazatel. Skryté vrstvy se nacházejí v privátní proměnné `deviceOutputsHiddenLayers`, což je další pole ukazatelů obsahující ukazatele odkazující na pole `float` hodnot. Třída `ParallelNeuralNet` hlavně slouží k udržování ukazatelů na pole neuronových sítí a jejich přepočítávání. Přehled všech atributů a metod můžeme vidět na obrázku 11.

Nejrozsáhlejší a pro popis nejdůležitější metody této třídy jsou v seznamu níže, u ostatních je popis uveden v dokumentaci nacházející se v elektronické příloze:

- `feedForward` – přijímá jako vstupní argument pole vstupních hodnot z trénovací nebo testovací sady. Stejně jako u sekvenční implementace je očekáváno, že jsme vstupní sadu



Obrázek 11: Třída ParallelNeuralNet.

předem normalizovali do intervalu shodného s oborem hodnot zvolené aktivační funkce. Přepočet sítě je stejný jako v sekvenční implementaci a v této metodě probíhá sekvenčně na CPU. Je určena pro vyhodnocení, resp. klasifikaci či aproximaci na naučených datech. Váhy sítě v paměti RAM jsou v privátní proměnné `hostWeights`.

- `parallelFeedForward` – určena pro vyhodnocování trénovací sady při učení. Zde přepočet probíhá tak, že je volán kernel podle počtu kopií neuronových sítí. Každé z těchto neuronových síti je předáno pole vstupních hodnot (jeden vzor). Jednotlivé neuronové sítě jsou spouštěny přes kernel `kernelParticlesFeedForward`. Výstupní hodnoty jsou uloženy do matice `deviceOutputs`.

- `setWeightsHost` nastaví váhy neuronové sítě do pole v paměti RAM přes generickou funkci `copyArray` v souboru *stdafx.h*.

Definice kernelů a funkcí určených pro spouštění v nich jsou spolu s definicemi metod třídy v souborech *ParallelNeuralNet.cuh* a *ParallelNeuralNet.cu*. Nachází se zde také deklarace ukazatele na funkce implementující aktivační funkce. Jejich popis je následujícím seznamu:

- `sigmoid` – funkce je napsána tak, aby mohla být vykonávána na CPU i v kernelu na GPU, je v ní implementován vztah 3 [1] funkce Sigmoid. Do kernelu je předávána jako argument.

- `hyperbolicTangens` – funkce je stavěna stejně jako ta předchozí, implementuje vztah 4 [3] funkce Hyperbolický Tangens.

- `kernelParticlesFeedForward` – kernel přepočítává podobně jako metoda `feedForward` vstupní vzor na neuronové síti.

5.5.6 Třída ParallelPSO

Instance této třídy obsahuje pole hodnot a ukazatelů tvořící paralelní implementaci algoritmu PSO. Přehled metod a privátních atributů je na obrázku 12.

Obrázek 12: Třída ParallelPSO.

PSO je v instanci vykonáván sekvenčně v iteracích. Paralelizován v blocích je výpočet rychlostí, pozic a nastavování nejlepších pozic, ve vláknech pak výsledné hodnoty účelové funkce všech částic. Algoritmus je pak rozdělen na několik paralelních částí, které jsou zpracovávány současně. Tyto části představuje většina metod třídy, zbytek metod spouští algoritmus PSO, učení sítě nebo vrací hodnoty instančních privátních proměnných. Všechny metody volající kernel funkci, mají implementováno řešení omezení počtu vláken v bloku popsané v podkapitole 5.5.4, nepracují-li pouze s jedním vláknem a jedním blokem. V souborech *ParallelPSO.cuh* a *ParallelPSO.cu* jsou také deklarovány a definovány testovací funkce, popsané v podkapitole 3.1.4

z [6]. Jednotlivé složitější metody jsou popsány v následujícím seznamu, první čtyři jsou určeny pro běh na CPU s tím, že volají paralelně navržené metody:

- `run` – metoda spouští test algoritmu PSO podle pseudokódu 1 [6]. Na začátku ze vstupního argumentu řetězce podmínkami získá ukazatel na účelovou funkci, pak volá metodu `init`. V cyklu o počtu iterací `iterations` volá metody `calculatePosition`, `calculateVelocity` a `calculateCostFunctionValue`. Přes `cudaMemcpy` z CUDA knihovny zkopíruje do paměti RAM výsledné hodnoty účelové funkce a `deviceBestGlobalValue`. Nakonec volá metodu `findBest`.

- `learnNeuralNet` – metoda funguje stejně jako předchozí zmíněná `run` podle pseudokódu 1 [6], ale místo volání účelové funkce volá metodu `particlesFeedForward` instance třídy `ParallelNeuralNet`. Předává jí ukazatel na pole vstupních hodnot. To je organizováno jako pole vstupů jednotlivých částic, umístěných v poli vzorů z trénovací sady. Nastavuje váhy sítě její metodou `setWeightsDevice`. Chyba neuronové sítě je pak vypočtena sekvenčně na CPU.

- `init` – inicializuje PSO, nastaví `hostBestGlobalValue` na maximální hodnotu datového typu `float` přes „class template“ `numeric_limits` C++ knihovny *limits* a překopíruje ji přes `cudaMemcpy` z CUDA knihovny do `deviceBestGlobalValue`. Volá sekvenčně na každou částici metody `setRandomPosition`, `setBestPosition` a `setRandomVelocity`. Nakonec volá metodu `setBestGlobalPosition` a nastavuje jí jako argument index první částice, resp. hodnotu 0.

- `findBest` – porovnává a nastavuje v cyklu podle počtu částic v proměnné `countParticles` hodnoty z `deviceParticlesCostFunctionValue` s `deviceParticlesBestValue` a `deviceBestGlobalValue` s `deviceParticlesBestValue`. Ověřuje tedy aktuálně nejlepší hodnoty s nově vypočtenými. V metodě je ověřování jednotlivých částic prováděno sekvenčně na CPU.

- `calculateCostFunction` – volá kernel `kernelCalculateCostFunctionValue`, nastavuje každou částici PSO jako jedno vlákno. Předává kernelu jako argument pole `deviceParticlesCostFunctionValue` pro výpočet výsledných hodnot účelové funkce, `deviceParticlesCurrentPosition` a ukazatel typu `CostFunction`. V kernelu proběhne pouze výpočet podle této funkce.

- `calculateVelocity` – metodou `generateRandomFloatArray` z instance v `randomGenerator` nechá vygenerovat pole náhodných čísel do `devicePregeneratedRandomArray` v cyklu, kde je počet iterací shodný s počtem částic. Přes kernel `kernelCopyArrayToMatrix` je překopíruje do `devicePregeneratedRandomMatrix`. Volá kernel `kernelCalculateVelocity` pro výpočet rychlosti podle vztahu 11 [6] a nastavuje pro každou částici blok. Pole `deviceParticlesCurrentVelocity` je zpracováváno ve vláknech. Pokud v kernelu dojde

k překročení meze, je na indexu aktuální částice v poli `deviceParticlesOutOfRange`, zjištěné podle indexu bloku, nastavena hodnota `true`. Metoda `calculateVelocity` nakonec volá metodu `checkParticles`.

- `calculatePosition` – volá kernel `kernelCalculatePosition` pro výpočet pozic podle vztahu 12 z [6], nastavuje pro každou částici blok. Pole `deviceParticlesCurrentPosition` je zpracováváno ve vláknech. V případě že dojde k překročení meze v některém z vláken, je stejně jako u předchozí metody nastavena hodnota na `true`. Pak volá metodu `checkParticles`.
- `setRandomPosition` – metoda volá metodu `generateRandomFloatArray` instance v `randomGenerator` a předává jí jako vstupní argument ukazatel na `devicePregenerateRandomNumbers`. Hodnoty tohoto pole přes kernel `kernelCopyArrayToMatrix` překopíruje do `deviceParticlesCurrentPosition`.
- `setRandomVelocity` – pracuje stejně jako předchozí metoda, jen jsou generovány náhodné hodnoty pro `deviceParticlesCurrentVelocity`.
- `setBestPosition` – volá kernel `kernelSetBestPosition`, předává mu pole `deviceParticlesCurrentPosition` a `deviceParticlesBestPosition`. Kernel překopíruje hodnoty z druhého pole do prvního podle zadané částice v `currentParticle`.
- `setBestGlobalPosition` – volá kernel `kernelSetBestGlobalPosition`, předává mu pole `deviceParticlesCurrentPosition` a `deviceBestGlobalPosition`. Kernel překopíruje hodnoty z matice `deviceParticlesCurrentPosition` do pole nejlepších globálních pozic podle indexu zadané částice v `currentParticle`. Počet bloků je nastaven na jeden.
- `checkParticles` – kontroluje instanční pole `deviceParticlesOutOfRange`, zda není některá částice mimo rozsah, resp. na hodnotu `true`. Je-li v poli tato hodnota, volá metodu `resetParticle` s indexem aktuální částice. Celá metoda probíhá na CPU.

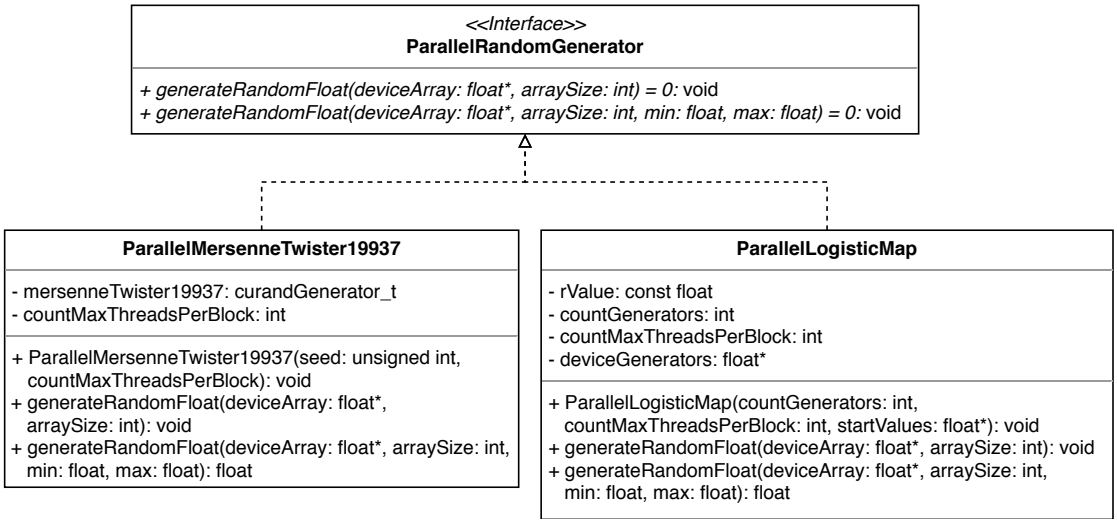
5.6 Generátory náhodných čísel

Třídy paralelních generátorů spadají stejně jako v sekvenční části do návrhového vzoru abstraktní továrna (angl. abstract factory). Obě třídy implementují rozhraní `ParallelRandomGenerator`. Generování je založeno na předem vygenerovaných polích náhodných `float` hodnot. Ty jsou pak předkládány kernelům.

Metody tříd, uvedených níže, přijímají na vstupu jako argument ukazatel na pole `float` hodnot v paměti zařízení (GPU). Do tohoto pole vygenerují metody `generateRandomFloatArray` náhodné float hodnoty. V případě třídy `ParallelLogisticMap` by se nabízela i možnost generovat náhodná čísla přímo v kernelu. Generátor `CURAND_RNG_PSEUDO_MT19937` je ale určen pro tzv. „Host API“ [10] a umožňuje paralelně generovat data pouze do pole v paměti GPU [20]. V knihovně můžeme najít i jiný Mersenne Twister generátor označovaný jako

CURAND_RNG_PSEUDO_MTGP32. Ten umožňuje generovat přímo v kernelu, bohužel však je omezen 256 vlákny na blok [20]. Z pohledu návrhu této paralelní CUDA části, který jsme si popsali v podkapitole 5.5.3, je pro tuto implementaci generátor nevyhovující.

Nastavení minimálních a maximálních hodnot, pokud není definována argumentem metody, jsou stejná jako u generátorů sekvenční implementace v podkapitole 5.4. U „seedu“ toto platí také, ale s rozdílem, že do konstruktoru třídy `ParallelLogisticMap` je předáváno pole počátečních náhodných `float` hodnot místo jedné hodnoty. Rozhraní a třídy můžeme vidět na obrázku 13.



Obrázek 13: Rozhraní ParallelRandomGenerator a třídy ParallelMersenneTwister19937 a ParallelLogisticMap.

Třída `MersenneTwister19937` slouží také jako „obalovací třída“ generátoru `CURAND_RNG_PSEUDO_MT19937` z knihovny *curand.h*. Oproti MT19937 [19] C++ knihovny *random* má jiné řazení [20]. Metoda `generateRandomFloatArray` volá funkci `curandGenerateUniform` knihovny *curand.h* a předává mu ukazatel pole, do kterého má generovat náhodná čísla. Protože tato funkce generuje náhodné hodnoty v rozsahu $(0, 1\rangle$ [20], metoda ověřuje, zda je vstupní argument `min` roven 0 a `max` roven 1. Pokud je splněna podmínka metoda končí. V případě, že není splněna, proběhne úprava hodnot v poli paralelně. Každou pozici v poli bude upravovat jedno vlákno. Pro tyto účely je definován kernel `kernelChangeIntervalFloatNumbers`.

Instance třídy `ParallelLogisticMap` ukládá generované hodnoty do pole privátní instanční proměnné `generators`. V konstruktoru jsou nastaveny startovací hodnoty. Zavoláním metody `generateRandomFloatArray` je volán kernel `kernelGenerateFloatNumbers`, který s pomocí funkce zařízení (device) `getRandomFloat` vygeneruje pole náhodných hodnot v zadaném rozsahu. Implementovaný vztah 14 logistické rovnice [9, 18] pak najdeme ve funkci `getRandomFloat`. Konstanta `rValue` je dle doporučení diplomové práce [18] inicializována na hodnotu $3,9999$.

6 Testování neuronové sítě a algoritmu rojení částic

Prioritní pro nás při testování implementace neuronové sítě bude měření doby učení s použitím generátorů náhodných čísel MT19937 Mersenne Twister [8, 19, 20] pro sekvenční aplikaci v knihovně jazyka C++ *random*, pro CUDA platformu v knihovně *curand.h* a logistické rovnice pro obě. Do měření není započteno čtení parametrů, trénovací a testovací sady vzorů. Měřeny budou pouze volané metody (paralelní i sekvenční) `learnNeuralNet` a `run`.

Každý test bude spuštěn ze statistického hlediska třicetkrát a časy zprůměrovány. Hodnota počtu opakování je v konstantní proměnné `COUNT_STATS_SAMPLES` v souboru *Parameters.h*, nastavena na hodnotu 10. Desetinná čísla jsou v konzoli i souboru zaokrouhleny na šest desetinných míst. Počet desetinných míst je definován v proměnné `FLOAT_PRECISION` v souboru *Parameters.h*. U testů neuronové sítě se objeví i průměr nejmenších chyb sítě po naučení.

Porovnávání výsledků na správnost či chybovost je založeno na výstupech souboru s výsledky testu. Při vyřešení klasifikačních úloh je ve výpisu do konzole a souboru zobrazován souhrn chybovosti jednotlivých testovaných vzorů, výpočet chybovosti vychází z podílu počtu špatných výsledků klasifikovaných vzorů a celkového počtu vzorů v sadě násobeno jedním-stem. Reálné výstupy jsou zaokrouhlovány v aplikacích na celé hodnoty, pokud je hodnota menší než $0,5$ zaokrouhlují dolů, nad $0,5$ nahoru. V případě aproximace jsou jen vypsány podrobnější výsledky do souboru.

Všechny výsledky testů uvedené v tabulkách, resp. časové a chybové hodnoty vychází ze souborů z výsledky vytvořených zkompilovanými 64. bitovými „release“ verzemi obou aplikací. Sekvenční implementace byla vytvořena jako jednovláknová aplikace, testy na ní byly spuštěny souběžně ve třech vláknech na CPU. Tyto soubory najdeme v elektronické příloze. Z důvodu snížení velikosti souboru a hlavně zlepšení přehlednosti byly odstraněny výpisy jednotlivých vah sítě každého testu.

6.1 Použitý software a hardware

Testování probíhalo pod operačním systému Windows 10 na desktopovém PC s čtyřjádrovým procesorem Intel i-5-7400 s 3,00 GHz (pro Turbo 3,50 GHz) a dedikovanou grafickou kartou NVIDIA GeForce GTX 1060 s celkem 1280 CUDA jádry a 6 GB VRAM GDDR5 paměti.

6.2 Normalizace metodou MIN-MAX

Jedná se o jednoduchou metodu, která upravuje hodnoty ze vstupní množiny do nového intervalu [21]. Normalizace byla provedena pomocí vztahu převzatého z článku [21]. Značení proměnných ve vztahu 15 se oproti originálu mírně liší, z důvodu lepší přehlednosti.

$$A = \left(\frac{A_i - A_{min}}{A_{max} - A_{min}} \right) \cdot (max - min) + min \tag{15}$$

kde,
A – množina hodnot k úpravě,
A_{min} – nejmenší hodnota v množině,
A_{max} – největší hodnota v množině.
max – největší hodnota nového intervalu.
min – nejmenší hodnota nového intervalu.

6.3 Test PSO

Tento test pouze ověřuje funkčnost samotného optimalizačního algoritmu. Je krátce shrnut v následujících několika odstavcích a proběhl na obou implementacích.

Nastavení parametrů vychází z doporučení v literatuře [6], tedy počet částic byl stanoven jako $10 \cdot dimenze$, maximální rychlost $V_{max} = \frac{1}{20} \cdot velikostRozsahu$ a rozsah podle konkrétní testovací účelové funkce. Test proběhl s nastavením dimenzí 2, 5, 20 na pěti testovacích funkcích popsaných v podkapitole 3.1.4 [6] a obou generátorech.

Sekvenční implementace algoritmu byla úspěšně otestována pro každý z implementovaných generátorů náhodných čísel, Mersenne Twister a logistickou rovnici. V každém testovém vzorku byl nalezen extrém při dimenzi 2 a 500 iteracích. U dimenzí 5 a 20 v případě Schwefelovy a 2. De Jongovy funkce se pozice alespoň přiblížili v řádu jednotek až desítek k extrému, přičemž iterace byly nastaveny pro 5 dimenzí na 5 až 8 tisíc iterací. Ostatní funkce se také podobně přibližovali k extrému již při 5000 iteracích.

Paralelní implementace byla také otestována pro oba generátory. Oproti sekvenční implementaci se při 2 dimenzích a 500 iteracích hodnoty přiblížili k extrému v řádu desetin a setin pro všechny De Jongovy funkce a Schwefelovu funkci. Při dalších dimenzích 5 a 20 a nastavených 5000 iteracích se přiblížili hodnoty v řádu jednotek a u 20 dimenzí v řádu desítek hodnot.

6.4 Problém XOR

Cílem tohoto problému [2] je naučit neuronovou síť exkluzivní disjunkci, její pravdivostní hodnoty můžeme vidět v tabulce 1. Soubory s trénovací a testovací sadou vzorů jsou pojmenovány *xor_train_set.csv* a *xor_test_set.csv* a najdeme je v elektronické příloze v adresáři *Testování*.

x1	**x2**	**x1 ⊕ x2**
0	0	0
0	1	1
1	0	1
1	1	0

Tabulka 1: Pravdivostní hodnoty exklusivní disjunkce (XOR) [2].

6.4.1 Výsledky klasifikace problému XOR na neuronové síti

Hodnoty parametrů `vMax` byly vypočteny podle $V_{max} = \frac{1}{20} \cdot velikostRozsahu$ a počet částic $2 \cdot Dimenze$, obojí doporučení pochází z literatury [6]. V případě nastavených 12 vah, je počet částic snížen na 20 z důvodu nižší doby učení při přibližně stejné chybě učení. Stanovení rozsahu je podle počtu částic a na základě lepších výsledků při testování. Ostatní hodnoty parametrů algoritmu PSO byly zvoleny na základě doporučení ze stejné literatury [6]. Testy byly provedeny pro 3000 iterací se dvěma generátory náhodných čísel MT19937 a logistickou rovnicí.

Použité parametry najdeme shrnuty v tabulce 2. Neuronové síti byly nastaveny dva vstupy a jeden výstup podle literatury [2], najdeme je v tabulce 1 [2]. Jako aktivační funkce byla použita funkce Sigmoid. Uvedené počty částic jsou stanoveny na základě vykazování lepších výsledků v průběhu testování. Vzhledem k tomu, že jsou obě sady (trénovací i testovací) stejné, bude shrnuta chybovost v klasifikaci XOR problému jednou procentuální hodnotou.

Parametr	**2 neurony (6 vah)**	**4 neurony (12 vah)**
`count_input`	2	2
`count_output`	1	1
`count_layer`	1	1
`count_particle`	12	20
`iterations`	3000	3000
`rangeMax`	6	10
`rangeMin`	-6	-10
`c1`	2,0	2,0
`c2`	2,0	2,0
`vMax`	0,6	1
`wStart`	0,9	0,9
`wEnd`	0,4	0,4

Tabulka 2: Jednotlivé parametry ze souboru pro různé počty neuronů ve skryté vrstvě XOR problému.

Při 1000 iterací byla chyba učení neuronové sítě oproti 3000 iterací o desetinné řády vyšší a docházelo ke chybné klasifikaci naučených vzorů. V tabulkách 3 a 4 se nachází srovnání časových hodnot a chyb pouze úspěšnějších výsledků (při 3000 iterací).

Z uvedené tabulky 2 vyplývá, že byl XOR problém testován na dvou typech nastavení parametrů v různém počtu vah sítě. Pro nastavené dva neurony (6 vah) v sekvenční implementaci s logistickou rovnicí dosahovali časové hodnoty v průměru $29,17$ milisekund, ale s chybou $3,75 \times 10^{-1}$, která způsobovala nepřesné výstupní hodnoty již v řádu desetin a průměrná chybovost klasifikace trénovací i testovací sady vzorů byla $44,17\%$. Při čtyřech neuronech (12 váhách) se průměrná doba učení zvýšila o $30,86$ milisekund na $60,03$ milisekund a průměrná chyba se snížila na hodnotu $1,52 \times 10^{-1}$. Chybovost při klasifikaci obou sad klesla na 0%.

Projekt	Váhy	Chyba (Avg)	Max	Avg	Med	Min
CPU	6	$3,75 \times 10^{-1}$	30,00	29,17	29,00	28,00
CUDA	6	$2,85 \times 10^{-1}$	61750,00	40460,00	39410,00	38700,00
CPU	12	$1,52 \times 10^{-1}$	62,00	60,03	60,00	59,00
CUDA	12	$2,88 \times 10^{-1}$	128114,00	123470,00	123430,00	120300,00

Tabulka 3: Doby učení sítě na XORu v milisekundách s průměrnou chybou učení pro 3000 iterací a s použitím logistické rovnice.

U generátoru MT19937 sekvenční implementace v tabulce 4 při dvou neuronech (6 váhách) dosahovala průměrná doba učení $51,03$ milisekund. Při čtyřech neuronech (12 váhách) stoupla průměrná doba o $86,74$ milisekund. Při klasifikaci klesla chybovost vyhodnocení u obou sad v rámci stejného generátoru z $47,50\%$ na 0%. Vzhledem k tomu, že se po čtyřech neuronech (12 váhách) v obou implementacích s dalšími přidanými neurony doba učení zvýšila, chyba učení neuronové sítě klesla ale chybovost klasifikace zůstala na 0%, nebyl do testu žádný další počet neuronů zahrnut.

Projekt	Váhy	Chyba (Avg)	Max	Avg	Med	Min
CPU	6	$2,80 \times 10^{-1}$	54,00	51,03	51,00	49,00
CUDA	6	$1,07 \times 10^{-2}$	70630,00	66200,00	66050,00	65080,00
CPU	12	$1,46 \times 10^{-2}$	273,00	137,77	123,00	119,00
CUDA	12	$1,77 \times 10^{-2}$	204500,00	197250,00	196560,00	194700,00

Tabulka 4: Doby učení sítě na XORu v milisekundách s průměrnou chybou učení pro 3000 iterací a s použitím Mersenne Twisteru.

V paralelní CUDA implementaci pro oba generátory došlo mezi 2 neurony (6 vah) a 4 neurony (12 vah) k přibližně trojnásobnému prodloužení doby učení. Průměrná chybovost klasifikace byla u 2 neuronů s logistickou rovnicí $44,17\%$ a u Mersenne Twisteru $44,67\%$, u 4 neuronů 0%. V obou tabulkách 3 a 4 lze vidět, že průměrná chyba učení obou implementací se liší pouze v řádech desetin či setin.

V tabulkách 3 a 4 můžeme také vidět větší hodnotu chyby neuronové sítě v řádech desetinných míst pro generátor Logistické rovnice než u Mersenne Twisteru u obou implementací.

6.5 Učení a aproximace funkce Sinus

Kromě klasifikačních problému si zde otestujeme i aproximaci funkce Sinus. Trénovací a testovací sada byla vytvořena pomocí vedlejšího projektu vytvořeného přímo pro tuto práci, nazvaného *SinusDataSetGenerator*. Ten vygeneroval dva soubory s trénovacími a testovacími vzory. Vstupy jsou generovány v radiánech v intervalu $\langle 0, 2\pi \rangle$ a předem normalizovány pro použití s aktivační funkcí hyperbolický tangens do intervalu $\langle -1, 1 \rangle$. Počet vzorků/vzorů je definován v proměnné `COUNT_SAMPLES` v souboru *SinusDataSetGenerator.cpp*. Jednotlivé vzory jsou generovány postupně od nejmenšího po největší úhel. Pro trénovací sadu jsou generovány náhodné hodnoty

ve stejném intervalu seřazené od nejmenšího úhlu po největší. V obou sadách jsou i očekávané výstupy.

6.5.1 Výsledky učení a aproximace sinus funkce na neuronové síti

Hodnoty v parametrech jsou pro PSO jako v případě XOR problému v podkapitole 6.4.1 nastaveny na stejné hodnoty. Počet iterací byl nastaven na 5000. Počet vstupů a výstupů byl nastaven na jeden a počet vrstev stejně, s ohledem na počet vzorů v trénovací sadě a předchozí testy. Testování bylo prováděno na dvou trénovacích sadách, jedna obsahuje 20 a druhá 40 vzorů.

Parametr	**Hodnota**
`count_input`	1
`count_output`	1
`count_layer`	2
`count_neuron_in_layer`	100, 20
`count_particle`	12
`iterations`	5000
`rangeMax`	6
`rangeMin`	-6
`c1`	2,0
`c2`	2,0
`vMax`	0,6
`wStart`	0,9
`wEnd`	0,4

Tabulka 5: Jednotlivé parametry ze souboru použité pro testování funkce sinus.

Ze všech testů byly vybrány pouze výsledky s nejlepšími hodnotami, průměrná chyba učení je v řádech desetin. Výstupy po vyhodnocení trénovací sady se lišili ve většině případů o několik setin až desetin u paralelní implementace, v případě sekvenční implementace častěji o desetiny. U testovací sady se pak výstupní hodnoty lišili v řádech desetin u obou implementací.

Projekt	**Vzorů**	**Váhy**	**Chyba (Avg)**	**Max**	**Avg**	**Med**	**Min**
CPU	20	2120	$7,68 \times 10^{-1}$	9,45	8,77	8,70	8,61
CUDA	20	2120	$3,37 \times 10^{-1}$	293,81	286,42	285,14	282,56
CPU	40	2120	$2,44 \times 10^{+0}$	17,37	15,91	15,78	15,42
CUDA	40	2120	$7,66 \times 10^{-1}$	457,27	454,49	454,27	451,29

Tabulka 6: Doby učení sítě na funkce sinus v sekundách s průměrnou chybou učení pro 5000 iterací a s použitím logistické rovnice.

V tabulkách 6 a 7 jsou shrnuty výsledky učení neuronové sítě na sekvenční a paralelní implementaci při 2120 vahách pro 20 a 40 vzorů trénovací sady. Z těchto tabulek lze také vyčíst, že chyba učení sítě na sekvenční i paralelní implementaci byla u Mersenne Twisteru nižší než

logistické rovnice. Průměrná doba učení na paralelní implementaci se zvyšovala mezi generátory přibližně o 150 sekund a mezi sadami cca o 175 sekund.

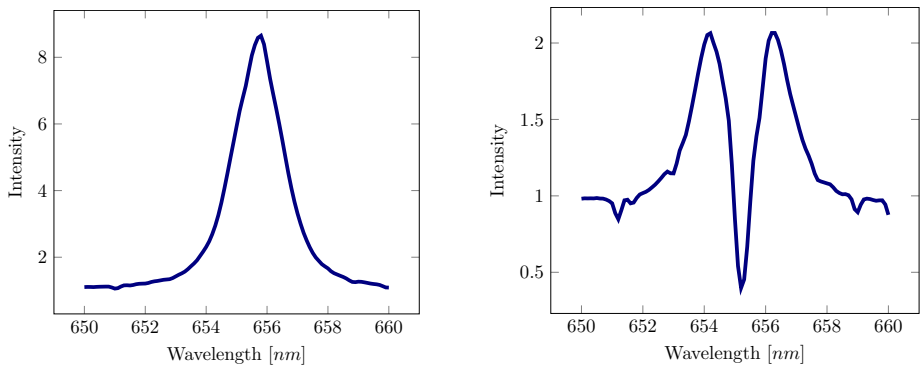
Projekt	Vzorů	Váhy	Chyba (Avg)	Max	Avg	Med	Min
CPU	20	2120	$2,32 \times 10^{-1}$	20,00	19,31	19,21	18,99
CUDA	20	2120	$2,11 \times 10^{-1}$	435,71	426,50	427,90	417,03
CPU	40	2120	$6,69 \times 10^{-1}$	27,55	26,19	26,04	25,82
CUDA	40	2120	$5,60 \times 10^{-1}$	623,46	613,76	608,99	605,87

Tabulka 7: Doby učení sítě na funkce sinus v sekundách s průměrnou chybou učení pro 5000 iterací a s použitím Mersenne Twisteru.

6.6 Vzorová testovací data

Pro další klasifikační testování neuronové sítě jsou použita astronomická „Stellar data“ [22], která pocházejí z Astronomického ústavu Akademie věd České Republiky v Ondřejově a byla poskytnuta vedoucím této práce. Tyto data jsou řazeny po řádcích a určují závislost intenzity světla na vlnové délce rozdělených do celkem čtyř tříd (v [22] označováno jako „Class“) [22].

Protože je celá sada příliš velká a s ohledem na výsledky z aproximace funkce sinus, bylo vybrány k naučení neuronové sítě pouze dvě třídy, první a čtvrtá. Tyto třídy v celém datové souboru reprezentuje v pořadí 178 a 55 vzorů a můžeme je vidět na skupině obrázků 14 převzaté z článku [22].



(a) Třída 1 (b) Třída 4

Obrázek 14: Obrázky 1. a 4. třídy převzaté z článku [22].

Úprava těchto dat je až na menší změny stejná jako v článku [22]. Z tříd od nalezeného vrcholu je odebráno 100 bodů, jelikož lze podle [22] okolí považovat za šum. Přesněji řečeno, od vrcholu zprava je vybráno 50 hodnot a zleva 49. Pro trénovací sadu bylo ručně a náhodně extrahováno 5% z počtu každé třídy, zprůměrováno a nakonec normalizováno do intervalu $\langle 0, 1 \rangle$ vztahem 15 metody MIN-MAX [21]. Interval byl zvolen podle použité aktivační funkce Sigmoid [1] již popsaném v podkapitole 2.3.2.1.V trénovací sadě se tedy nachází jeden vzor. Do testovací sady

bylo vybráno náhodně šest vzorů, od každé třídy tři. Jejich úprava byla stejná jako u trénovací sady, s výjimkou průměrování, které nebylo prováděno.

6.6.1 Výsledky učení a klasifikace vzorových dat na neuronové síti

Parametry `vMax`, `c1`, `c2`, `wStart` a `wEnd` byly nastaveny na hodnoty stejně jako při testování XOR problému v podkapitole 6.4.1. Počet iterací je 5000. V případě nižších hodnot, byla chyba o řády desetinných míst větší. Počet částic byl stanoven na základě testování a rozsah podle počtu částic. Hodnoty parametrů jsou shrnuty v tabulce 8. Tabulky 9 a 10 obsahují průměrnou chybu učení neuronové sítě a časové údaje o učení. Počet vrstev a neuronů v nich byl nastaven odhadem vůči trénovacím vzorům a počtu vstupů. U těchto výsledných hodnoty byla zvolena jen jedna vrstva po 25 a také po 50 neuronech.

Parametr	**Hodnota**
`count_input`	100
`count_output`	2
`count_layer`	1
`count_particle`	12
`iterations`	5000
`rangeMax`	6
`rangeMin`	-6
`c1`	2,0
`c2`	2,0
`vMax`	0,6
`wStart`	0,9
`wEnd`	0,4

Tabulka 8: Jednotlivé parametry ze souboru použité pro testování vzorových dat.

Při učení na sekvenční implementaci s generátorem logistické rovnice v tabulce 9 průměrná chyba učení klesla z $6,14 \times 10^{-3}$ při 2550 vahách na $1,12 \times 10^{-10}$ při 5100 vahách a průměrná doba se přibližně zvýšila o přibližně dvojnásobek. Klasifikace testovací sady při 6 vzorech skončila s průměrnou chybovostí při 25 neuronech (2550 vah) na 50,47%, při 50 neuronech pak na 0%.

Projekt	**Neurony**	**Váhy**	**Chyba (Avg)**	**Max**	**Avg**	**Med**	**Min**
CPU	25	2550	$6,14 \times 10^{-3}$	2,61	2,54	2,52	2,46
CUDA	25	2550	$1,15 \times 10^{-10}$	126,09	116,48	120,72	12,39
CPU	50	5100	$1,12 \times 10^{-10}$	5,18	5,05	5,05	4,98
CUDA	50	5100	$1,09 \times 10^{-16}$	186,18	178,23	177,57	174,58

Tabulka 9: Doby učení sítě na vzorových datech v sekundách s průměrnou chybou učení pro 5000 iterací a s použitím logistické rovnice.

U generátoru MT19937 v tabulce 10 si můžeme všimnout, že oproti logistické rovnici došlo k výraznému snížení průměrné chyby učení. Průměrná doba při 25 neuronech (2550 vahách),

se ale téměř trojnásobně zvýšila a v případě 50 neuronů (5100 vah) se ještě oproti 25 neuronů přibližně dvojnásobně zvýšila. V klasifikaci testovací sady (6 vzorů) byla průměrná chybovost pro 25 neuronů na $4,93\%$ a pro 50 na $7,03\%$. Průměrná chybovost klasifikace trénovací sady (2 vzory) byla ve všech případech na sekvenční implementaci při 0%

Projekt	**Neurony**	**Váhy**	**Chyba (Avg)**	**Max**	**Avg**	**Med**	**Min**
CPU	25	2550	$8,39 \times 10^{-6}$	15,31	15,07	15,04	14,98
CUDA	25	2550	$3,19 \times 10^{-9}$	260,80	254,79	254,04	253,04
CPU	50	5100	$0,28 \times 10^{-12}$	30,43	30,11	30,09	29,90
CUDA	50	5100	$2,13 \times 10^{-12}$	368,72	311,74	306,52	300,16

Tabulka 10: Doby učení sítě na vzorových datech v sekundách s průměrnou chybou učení pro 5000 iterací a s použitím Mersenne Twisteru.

V tabulkách 9 a 10 můžeme vidět, že v paralelní implementaci se průměrná doba učení s Mersenne Twisterem oproti logistické rovnici zvýšila téměř o dvojnásobek. U logistické rovnice na CUDA platformě při zdvojnásobení počtu vah v neuronové síti se průměrná doba učení zvýšila „pouze“ o $61,75$ sekund (z $116,48$ sekund na $178,23$ sekund) v poměru oproti sekvenční implementaci jejíž doba učení téměř dvojnásobná. Průměrná chyba učení sítě pro implementaci na CUDA platformě je horší u 50 neuronů (5100 vah) než 25 (2550 vah), u CPU je to naopak. Průměrná chyba při klasifikaci trénovací sady (2 vzorů) byla pro oba generátory a všechny váhy 0%. Testovací sada (6 vzorů) měla chybovost pro 25 neuronů $2,20\%$ a pro 50 neuronů $4,33\%$.

V případě Mersenne Twisteru je situace podobná, jen se průměrná doba učení mezi 25 a 50 neurony v poměru zvýšila o $56,95$ sekund, na CPU se doba zdvojnásobila stejně jako u logistické rovnice. Dále můžeme vidět v tabulce 10 průměrnou chybu učení neuronové sítě, která je nejlepší u 50 neuronů. Chybovost při klasifikaci testovací sady (6 vzorů) byla pro 2550 vah $2,17\%$, u 5100 vah $1,33\%$. Trénovací sada měla chybovost klasifikace na 0%.

I při řešení tohoto problému měl algoritmus rojení částic menší čas učení než s použitím logistické rovnice než Mersenne Twisteru ale s horší chybou učení.

7 Závěr

Hlavním cílem této bakalářské práce bylo implementovat neuronovou síť v jazyce C++ sekvenčně a následně paralelně na CUDA platformě. Jak již bylo zmíněno v úvodu 1, po domluvě z vedoucím práce byla zvolena implementace vícevrstvé dopředné sítě. Pro učení této sítě byl implementován evoluční algoritmus rojení částic.

Na začátku první části této práce byla nastíněna problematika umělých neuronových sítí, jejich struktura, typy a různé metody učení. Dalším bodem byly evoluční optimalizační techniky (nebo také algoritmy) s vysvětlením základních pojmů jako jsou populace, jedinec, účelová funkce, testovací funkce. Popis parametrů a průběh algoritmu rojení částic. Tato část byla zakončena popisem hardwarové architektury a programátorského modelu paralelní platformy CUDA na grafických kartách společnosti NVIDIA.

Ve druhé části jsme si popsali uživatelské rozhraní a generátory náhodných čísel, tedy zařazení Mersenne Twister do implementace a logistickou rovnici. Dále objektově orientovanou sekvenční implementaci neuronové sítě a algoritmu PSO a také statické třídy s aktivačními funkcemi neuronu a testovacími účelovými funkcemi. Nakonec jsme si popsali návrh, třídy a syntakticky nejrozsáhlejší funkce paralelní CUDA implementace neuronové sítě a algoritmu jejího učení.

V kapitole testování jsme mohli vidět testy tří úloh, dvou klasifikačních a jedné aproximační. V případě klasifikace se jednalo a úlohy jako XOR problém a klasifikace vzorových dat – Spekter, které byly v případě volby většího počtu vah neuronové sítě se schopny naučit vzorová data, jak naučená, tak i podobná na obou implementacích. Aproximační úlohou byla aproximace funkce sinus při níž byla testována na dvou trénovacích sadách, jedná měla 20 vzorů a druhá 40. Při aktuální volbě hodnot parametrů PSO, počtu vrstev a neuronů neuronové sítě se výstupy lišili v řádech hodnot desetin a setin. Ve všech případech byla paralelní implementace pomalejší než sekvenční. Osobně si myslím, že je to způsobeno přesuny dat mezi pamětí hosta (RAM) a pamětí zařízení (GPU) a alokací pomocných polí v paměti GPU. Z tabulek 3, 4, 6, 7, 9 a 10 lze usoudit, že učení s použitím logistické rovnice bylo rychlejší než Mersenne Twisteru, ale za cenu vyšší chyby učení neuronové sítě.

V této bakalářské práci jsem napsal zdrojový kód sekvenční a paralelní implementace neuronové sítě a algoritmu rojení částic pro jejich učení na základě popisu principů fungování z literárních publikací, vědeckých článků a dalších uvedených v seznamu použité literatury. Také jsem se pokusil jednoduše navrhnout a paralelizovat neuronovou síť a algoritmus rojení částic na CUDA platformě a ošetřit omezení počtu vláken na blok pro potřeby této implementace.

Domnívám se, že tato práce by mohla být v budoucnu rozšířena o případné učení s pomocí dalších evolučních algoritmů, pro zajímavost lze zmínit např. použití diferenciální evoluce nebo algoritmus SOMA. S tím souvisí i použití různých generátorů pseudonáhodných čísel jako např. kromě již v této práci použitého Mersenner Twisteru a logistické funkce také použití různých chaotických generátorů čísel. Jednou z dalších možností je i zakomponování této práce, resp.

implementace do nějakého řešeného problému, kde by implementace této práce sloužila k jeho řešení.

Literatura

[1] VONDRÁK, Ivo. *Umělá inteligence a neuronové sítě*. 3. vyd. Ostrava: VŠB – TECHNICKÁ UNIVERZITA OSTRAVA, 2009, 140 s. ISBN 978-80-248-1981-5.

[2] BEALE, Russell a Tom JACKSON. *Neural Computing-an introduction*. CRC Press, 1990. ISBN 0-85274-262-2.

[3] OLGAC, A. Vehbi a Bekir KARLIK. Performance analysis of various activation functions in generalized MLP architectures of neural networks. *International Journal of Artificial Intelligence and Expert Systems* [online]. 2011, **1**(4), s. 111–122 [cit. 2019-04-04]. Dostupné z: `https://www.researchgate.net/publication/228813985_Performance_Analysis_of_Various_Activation_Functions_in_Generalized_MLP_Architectures_of_Neural_Networks`

[4] GLOROT, Xavier a Yoshua BENGIO. Understanding the difficulty of training deep feedforward neural networks. In: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics* [online]. PLMR, 2010, **9**, s. 249–256 [cit. 2019-04-04]. Dostupné z: `http://proceedings.mlr.press/v9/glorot10a.html`

[5] MENDES, Rui, Paulo CORTEZ, Miguel ROCHA a José NEVES. Particle swarms for feedforward neural network training. In: *Proceedings of the 2002 International Joint Conference on Neural Networks. IJCNN'02 (Cat. No.02CH37290)* [online]. 2002, **2**, s. 1895–1899 [cit. 2019-04-24]. DOI: 10.1109/IJCNN.2002.1007808. ISSN 1098-7576. Dostupné z: `https://ieeexplore.ieee.org/abstract/document/1007808`

[6] ZELINKA, Ivan, Zuzana OPLATKOVÁ, Miloš ŠEDA, Pavel OŠMERA a František VČELAŘ. *Evoluční výpočetní techniky: principy a aplikace*. 1.vyd. Praha: BEN, 2009, 534 s. ISBN 978-80-7300-218-3.

[7] KENNEDY, James, Russell C. EBERHART a Yuhui SHI. *Swarm intelligence*. San Francisco: Morgan Kaufmann, 2001, 512 s. ISBN 15-586-0595-9.

[8] MATSUMOTO, Makoto a Takuji NISHIMURA. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation* [online]. 1998, **8**(1), 3-30 [cit. 2019-03-20]. DOI: 10.1145/272991.272995. ISSN 1049-3301. Dostupné z: `http://doi.acm.org/10.1145/272991.272995`

[9] MAY, Robert M. *Stability and complexity in model ecosystems*. 1. vyd. Princeton: Princeton University Press, 2001, 265 s. ISBN 06-910-8861-6.

[10] NVIDIA CORPORATION. CUDA C PROGRAMMING GUIDE: *Design Guide* [online]. Nvidia Corporation, ©2019, v10.1 [cit. 2019-04-01]. Dostupné z: https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

[11] NVIDIA CORPORATION. NVIDIA CUDA Compute Unified Device Architecture: *Programming Guide* [online]. Nvidia Corporation, ©2007, v1.0 [cit. 2019-04-01]. Dostupné z: http://developer.download.nvidia.com/compute/cuda/1.0/NVIDIA_CUDA_Programming_Guide_1.0.pdf

[12] JIA, Yangqing, Evan SHELHAMER, Jeff DONAHUE, Sergey KARAYEV, Jonathan LONG, Ross GIRSHICK, Sergio GUADARRAMA a Trevor DARRELL. Caffe: Convolutional Architecture for Fast Feature Embedding. *ArXiv preprint arXiv:1408.5093* [online]. 2014 [cit. 2019-03-16]. Dostupné z: https://arxiv.org/pdf/1408.5093.pdf

[13] CHETLUR, Sharan, Cliff WOOLLEY, Philippe VANDERMERSCH, Jonathan COHEN, John TRAN, Bryan CATANZARO a Evan SHELHAMER. Cudnn: Efficient primitives for deep learning. *ArXiv preprint arXiv:1410.0759v3* [online]. 2014 [cit. 2019-03-16]. Dostupné z: https://arxiv.org/pdf/1410.0759v3.pdf

[14] ABADI, Martín, Paul BARHAM, Jianmin CHEN, Zhifeng CHEN, Andy DAVIS, Jeffrey DEAN, Matthieu DEVIN, Sanjay GHEMAWAT, Geoffrey IRVING, Michael ISARD, Manjunath KUDLUR, Josh LEVENBERG, Rajat MONGA, Sherry MOORE, Derek G. MURRAY, Benoit STEINER, Paul TUCKER, Vijay VASUDEVAN, Pete WARDEN, Martin WICKE, Yuan YU a Xiaoqiang ZHENG. TensorFlow: A System for Large-Scale Machine Learning. In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 2016, s. 265–283 [cit. 2019-04-04]. ISBN: 978-1-931971-33-1. Dostupné z: https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf

[15] SIERRA-CANTO, Xavier, Francisco MADERA-RAMIREZ a Victor UC-CETINA. Parallel Training of a Back-Propagation Neural Network Using CUDA. In: *2010 Ninth International Conference on Machine Learning and Applications* [online]. IEEE, 2010, s. 307-312 [cit. 2019-04-04]. DOI: 10.1109/ICMLA.2010.52. Dostupné z: https://ieeexplore.ieee.org/abstract/document/5708849

[16] JANG, Honghoon, Anjin PARK a Keechul JUNG. Neural Network Implementation Using CUDA and OpenMP. In: *2008 Digital Image Computing: Techniques and Applications* [online]. IEEE, 2008. s. 155-161 [cit. 2019-04-05]. DOI: 10.1109/DICTA.2008.82. Dostupné z: https://ieeexplore.ieee.org/abstract/document/4700015

[17] KRIZHEVSKY, Alex, Ilya SUTSKEVER a Geoffrey E. HINTON. ImageNet Classification with Deep Convolutional Neural Networks. In: *Advances in*

Neural Information Processing Systems 25 [online]. Curran Associates, 2012, s. 1097–1105 [cit. 2019-04-06]. Dostupné z: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

[18] KOJECKÝ, Lumír. Analytické programování v jazyce C#. Ostrava, 2014. Diplomová práce (Ing.). Vysoká škola Báňská – Technická univerzita Ostrava, Fakulta elektrotechniky a informatiky, Katedra Informatiky, 2014. Vedoucí diplomové práce Prof. Ing. Ivan Zelinka, Ph.D.

[19] MICROSOFT CORPORATION. <random>. Microsoft Docs [online]. ©2017 [cit. 2019-03-22]. Dostupné z: https://docs.microsoft.com/en-us/cpp/standard-library/random

[20] NVIDIA CORPORATION. CURAND LIBRARY: *Programming Guide* [online]. Nvidia Corporation, ©2019 [cit. 2019-04-01]. Dostupné z: https://docs.nvidia.com/cuda/pdf/CURAND_Library.pdf

[21] PATOR, S. Gopal Krishna a Kishore Kumar SAHU. Normalization: A Preprocessing Stage. *ArXiv preprint arXiv:1503.06462* [online]. 2015 [cit. 2019-04-10]. Dostupné z: https://arxiv.org/ftp/arxiv/papers/1503/1503.06462.pdf

[22] KOJECKÝ, Lumír, Ivan ZELINKA a Petr ŠALOUN. Evolutionary synthesis of automatic classification on astroinformatic big data. *International Journal of Parallel, Emergent and Distributed Systems*. Taylor & Francis, 2017, **32**(5), s. 429–447. DOI: 10.1080/17445760.2016.1194984.

A Popis parametrů třídy Parameters

Zápis parametrů v souboru se vyskytuje v následujícím formátu: *název parametru=hodnota parametru*. Jedinou výjimkou je atribut `count_neuron_in_layer`, který má jako hodnotu parametru pole nenulových přirozených čísel o minimálně jedné hodnotě. Pokud se v parametru vyskytuje více hodnot, jsou odděleny čárkou. V tabulce níže jsou dále popsány názvy parametrů, typ jejich hodnot a omezení případných číselných hodnot, u kterých platí, že N je nenulové přirozené číslo do maximální velikosti stanovené datovým typem `Integer`.

Název parametru	**Dat. typ**	**Omezení**	**Popis**
run_pso_test	`bool`		Určuje zda má být spuštěn pouze test PSO, spustí se při hodnotě 1.
csv_separator	`char`		Oddělovač hodnot v CSV souboru.
train_set_file_path	`string`		Cesta s názvem k souboru s trénovacími daty.
test_set_file_path	`string`		Cesta s názvem k souboru s testovacími daty.
count_train_set_samples	`int`	1 .. N	Počet vzorů trénovací sady.
count_test_set_samples	`int`	1 .. N	Počet vzorů testovací sady.
count_inputs	`int`	1 .. N	Počet vstupů sítě.
count_ouputs	`int`	1 .. N	Počet výstupů sítě.
count_layer	`int`	1 .. N	Počet vrstev v síti.
count_neuron_in_layer	`int*` (pole)	1 .. N	Počet neuronů v jednotlivých vrstvách.
count_particles	`int`	1 .. N	Počet částic v populaci.
iterations	`int`	1 .. N	Počet iterací PSO.
dimension	`int`	1 .. N	Počet dimenzí PSO.
c1	`float`		Učící faktor.
c2	`float`		Učící faktor.
vMax	`float`	0 .. N	Max. rychlost částice.
wStart	`float`		Počáteční hodnota setrvačnosti.
wEnd	`float`		Koncová hodnota setrvačnosti.
rangeMin	`float`		Min. hodnota prostoru částice.
rangeMax	`float`		Max. hodnota prostoru částice.
activation_function	`string`		Aktivační funkce neuronu.
test_function	`string`		Testovací účelová funkce PSO.
random_generator	`string`		Generátor náhodných čísel.
evaluation_type	`string`		Typ vyhodnocení testovaných dat (`classify` nebo `aproximate`).

Tabulka 11: Tabulka názvů parametrů, typu hodnot s omezeními a jejich popis.

Parametry `activation_function`, `test_function` a `random_generator` mají definované identifikátory. Jejich popis pro `activation_function` můžeme vidět v tabulce 12, pro `test_function` v tabulce 13 a `random_generator` v tabulce 14

Funkce	**Identifikátor**
Sigmoid	`sigmoid`
Hyperbolický Tangens	`tanh`, `hyperbolicky_tangens` nebo `hyperbolic_tangens`

Tabulka 12: Tabulka identifikátorů aktivačních funkcí.

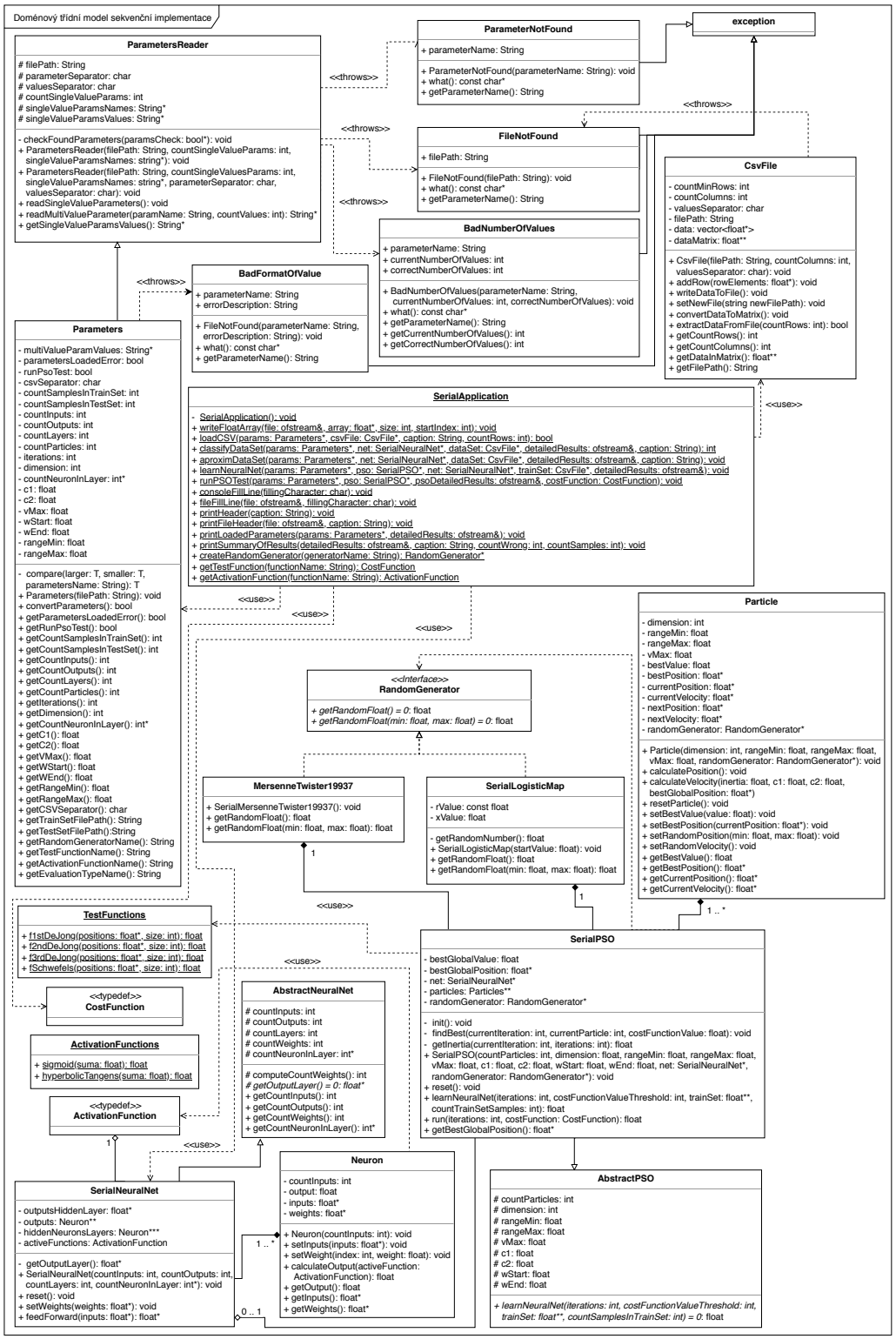
Funkce	**Identifikátor**
1st DeJong	`1_de_jongova_funkce` nebo `1st_de_jong`
2nd DeJong	`2_de_jongova_funkce` nebo `2nd_de_jong`
3rd DeJong	`3_de_jongova_funkce` nebo `3rd_de_jong`
4th DeJong	`4_de_jongova_funkce` nebo `4th_de_jong`
Schwefelova	`schwefelova_funkce` nebo `schwefels_function`

Tabulka 13: Tabulka identifikátorů testovacích účelových funkcí.

Generátor	**Identifikátor**
MT19937 – Mersenne Twister	`mt19937` nebo `mersenne_twister`
Logistická rovnice	`logisticka_rovnice` nebo `logistic_map`

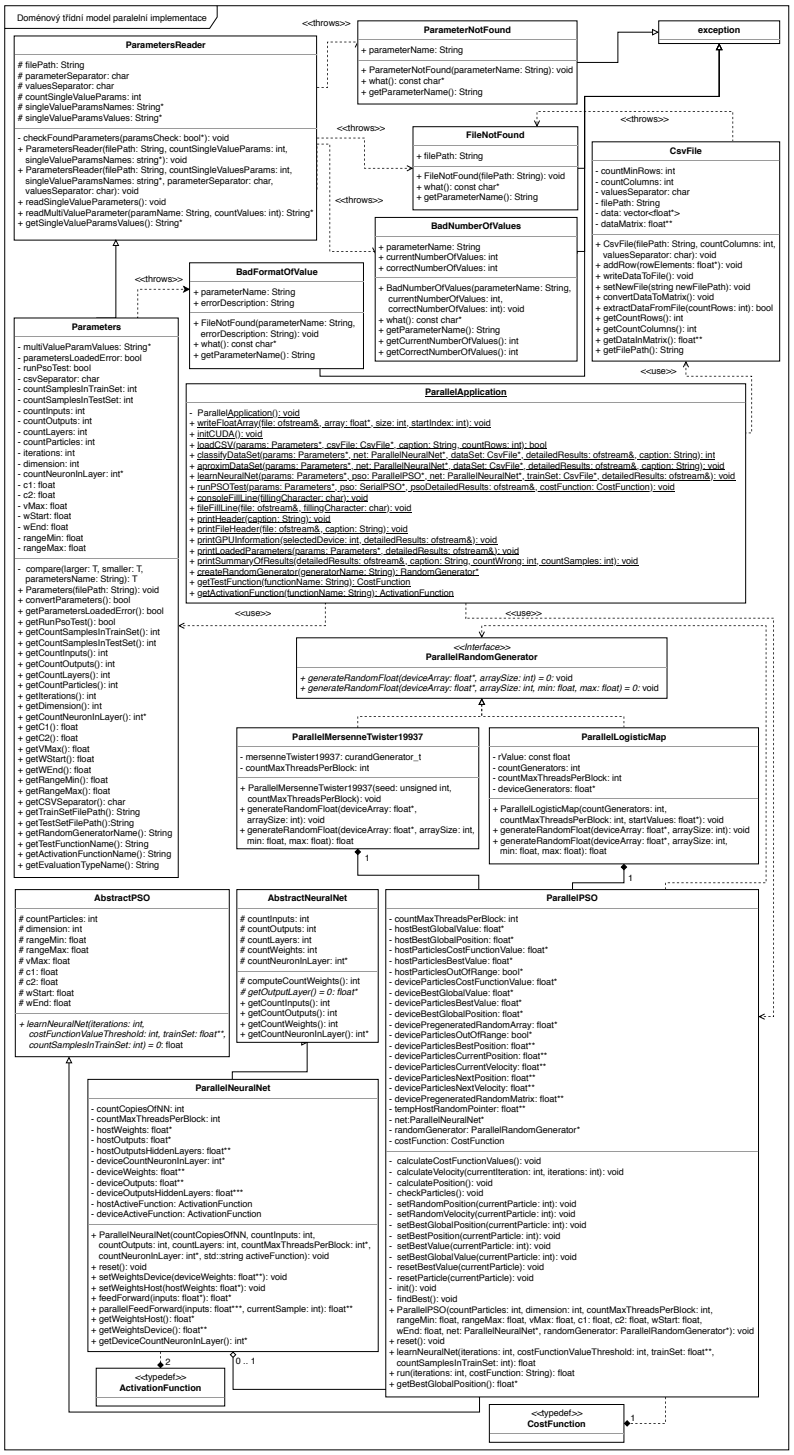
Tabulka 14: Tabulka identifikátorů generátorů náhodných čísel.

B Třídní diagram sekvenční implementace



Obrázek 15: Třídní diagram sekvenční implementace.

C Třídní diagram paralelní implementace



Obrázek 16: Třídní diagram paralelní implementace.